November 2019

# Managing Overheads in Asynchronous Many-Task Runtime Systems

Bibek Wagle
*Louisiana State University and Agricultural and Mechanical College*

# MANAGING OVERHEADS IN ASYNCHRONOUS MANY-TASK RUNTIME SYSTEMS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science and Engineering

by
Bibek Wagle
B.E., Tribhuvan University, 2008
M.S., Teesside University, 2011
December 2019

## Acknowledgments

First of all, heartfelt gratitude goes to my advisor Dr. Hartmut Kaiser, without whose support and invaluable guidance this work would not have been produced in this shape. I am also thankful to Dr. Bijaya B. Karki for being in my committee and for his support throughout my time at LSU. I am also thankful to my committee members Dr. Konstantin Busch and Dr. Jianhua Chen for their guidance and support.

I would like to thank Adrian for supporting me throughout the time I spent at LSU. I am greatly thankful for the time you have spent painstakingly proofreading all my work. I would also like to thank my friends at LSU for all their constructive discussions and valuable suggestions and also other researchers with whom over the course of this study I had the opportunity to collaborate with.

My gratefulness goes to my parents, Bimal and Sarita Wagle, who have always supported all my academic endeavors and for making all of this possible. A special thanks to my wife, Sona, for all the love and support and for always standing by me throughout this journey. I would like to thank my sister, Samjhauta and my brother in law Arun, for their support throughout this journey.

## Table of Contents

## List of Tables

## List of Figures

## Abstract

Asynchronous Many-Task (AMT) runtime systems are based on the idea of dividing an algorithm into small units of work, known as tasks. The runtime system is then responsible for scheduling and executing these tasks in an efficient manner by taking into account the resources provided to it and the associated data dependencies between the tasks. One of the primary challenges faced by AMTs is managing such fine-grained parallelism and the overheads associated with creating, scheduling and executing tasks. This work develops methodologies for assessing and managing overheads associated with fine-grained task execution in HPX, our exemplar Asynchronous Many-Task runtime system. Known optimization techniques, viz. active message coalescing, task inlining and parallel loop iteration chunking are applied to HPX. Active message coalescing, where messages bound to the same destination are aggregated into a single message, is presented as a solution to minimize overheads associated with fine-grained communications. Methodologies and metrics for analyzing fine-grained communication overheads are developed. The metrics identified and implemented in this research aid in evaluating network efficiency by giving us an intrinsic view of the underlying network overhead that would be difficult to measure using conventional methods. Task inlining, a method that allows runtime systems to manage the overheads introduced by a large number of tasks by merging tasks together into one thread of execution, is presented as a technique for minimizing fine-grained task overheads. A runtime policy that dynamically decides whether to inline a task is developed and evaluated on different processor architectures. A methodology to derive a largely machine independent constant that allows controlling task granularity is developed. Finally, the machine independent constant derived in the context of task inlining is applied to chunking of parallel loop iterations, which confirms its applicability to reduce overheads, in the context of finding the optimal chunk size of the combined loop iterations.

# Chapter 1. Introduction

The breakdown of Dennard scaling [1] and slowdown in Moore's Law [2] has resulted in a paradigm shift from the uni-processor era towards multi-core and many-core technologies. Following this shift in industry, today's supercomputers rely on many-core machines and hardware accelerators to achieve the FLOPS (Floating Point Operations Per Second) advertised. Hardware accelerator options such as GPUs and Xeon Phis increase the core count by orders of magnitudes. It is evident from the trends seen in table 1.1 that future machines will continue to increase intra-node concurrency via the addition of cores and accelerators. Keeping in line with the changes in the hardware, the focus of scientific software development is changing from relying on an increase in the clock speed of newer processors to exploiting parallelism from these new highly concurrent architectures.

Many modern HPC(High Performance Computing) applications use a hybrid programming model where MPI [3] is responsible for inter-node operations whereas another threading library such as OpenMP [4] is responsible for intra-node parallelism. MPI, which is an abbreviation for Message Passing Interface, is a widely used standard for distributed information exchange. First released in 1994, MPI is an example of SPMD (single program multiple data) parallelism where each node in the distributed architecture executes its own copy of the application and communicates with other nodes via message passing. MPI initially only supported synchronous messages which was later extended to support for sending asynchronous messages with the release of version 2 of the MPI standard. OpenMP, an abbreviation for Open Multi-Processing, is a standard for shared memory multiprocessing. OpenMP employs a fork-join method of parallelism where a master thread forks a number

Table 1.1. TOP500 by Year

| **Year** | **2005** | **2010** | **2015** | **2019** |
|---|---|---|---|---|
| **Machine** | BlueGene/L | Tianhe-1A | Tiahhe-2 | Summit |
| **Number of Cores** | 131,072 | 186,368 | 3,120,000 | 2,397,824 |
| **Number of Nodes** | 65,536 | 7,168 | 16,000 | 4,356 |

of slave threads in order to perform parallel tasks, at the end of which the slave threads join with the master thread. Since version 3 of OpenMP standard, support for asynchronous tasks have been added to OpenMP. Furthermore, OpenMP 4 provided directives to offload computation to accelerators. Scientific software development for HPC largely follows the MPI+X model where MPI is paired with some form of shared memory parallelism such as OpenMP, C++ threads, Pthreads or even hardware accelerators such as GPUs.

Asynchronous Many-Task(AMT) runtime systems have been gaining popularity in recent years as a possible solution towards effective utilization of available concurrency [5]. These runtime systems are founded on the idea of decomposing an algorithm into units of work, known as *tasks*, and executing them asynchronously. The amount of work contained in a task determines the *granularity* of the task. A task can be *fine-grained* containing only a few instructions or *coarse-grained* containing many instructions. The granularity of tasks plays a vital role in efficient utilization of hardware resources. Fine-grained tasks allow the total computation to be distributed evenly among the processors which enables better load balancing. In the event of an unforeseen delay in execution of a task, fine-grained tasks are preferable as the amount of work available in the system is abundant so that the resources can stay busy whereas a coarse-grained program experiencing the same delay will not be able to keep all of its resources busy and therefore stall the program execution. Fine-grained tasks also allow for flexibility in managing latencies. For example, a fine-grained tasks can be scheduled during the time another task is waiting for a resource to be ready. On the other hand, larger task granularity would not be able to effectively fill in the small gaps in CPU utilization due to lack of tasks small enough to execute during such period. Hence, fine-grained parallelism exposed by Asynchronous Many-Task runtimes enables effective load balancing and latency management that has the potential for better system utilization [6–8].

The benefits of employing fine-grained tasks can be nullified by the overheads associated with the creation and management of these tasks. Each task has an overhead cost

associated with it that can add up to significant portion of the overall computation time. Overheads are defined as excess work that needs to be carried out in order to perform actual computation. Overheads can also be thought of as the cost of parallelization or the cost that would not exist if the same application was run serially. The granularity of a task is an important factor when talking about overheads of the task. For example, if the granularity of the task is small, the overheads associated with the task may be comparable to the amount of work performed by the task. In such a case, large portion of computational time is spend on overheads. Conversely, in the case of coarse-grained tasks, the overheads may be a small fraction of the overall computation contained in the task. However, as the coarseness of the task increases, parallelism is negatively impacted. The key to deal with the overheads is to amortize the cost of overheads with useful computation. Therefore, the amount of work performed by a task should be large enough such than the overheads of creating and managing the task itself does not account for significant amount of computational time. Since a larger task may results in lower utilization whereas smaller tasks may result in overheads accounting for significant amount of overall application time, there needs to be a delicate balance between the granularity of the task and the amount of parallelism in the system. In order to efficiently utilize today's highly concurrent systems, effective management of overhead costs of fine-grained tasks is the key.

Overheads in the case of Asynchronous Many-Task runtimes can be broadly classified into two categories: those that pertain to creation and management of tasks that are executed locally in the node where the task was created, and those that pertain to tasks that are executed in a node different from the one where the task was created. In the context of this work, we refer to the two categories of tasks as locally executed tasks and remotely executed tasks. Overheads incurred by locally executed tasks arise from creation and management of these tasks. In the context of remotely executed tasks, additional overheads specific to remote execution must also be accounted for such as converting the task into an active message, serialization, transporting to the destination, de-serialization

and recreation of the original task at the destination.

This work provides methodologies for assessing and managing overheads associated with fine grained task execution in HPX, our exemplar asynchronous many task runtime system. Known optimization techniques, viz. active message coalescing, task inlining and parallel loop iteration chunking are applied to HPX. In the context of remotely executed tasks, active message coalescing is presented as a means to improve application performance. Methodologies and metrics for analyzing the overheads associated with transmission and reception of active messages in the context of HPX is developed. With regards to locally executed tasks, task inlining, where a child task is executed by the parent, is explored. A dynamic policy that decide whether to inline a particular task based on profiling information is also presented. Methodologies for determining a largely machine independent constant , $\lambda_{min}$, that allows controlling the granularity of tasks is also presented. $\lambda_{min}$ allows establishing the lower bound on the size of the task and denotes the point where the effects of overheads have been amortized. Furthermore, chunking of parallel loop iterations is applied in the context of HPX. Existing policy for automatically chunking parallel loops in HPX is extended to use $\lambda_{min}$ derived in the context of task inlining.

### 1.1. Research Contributions

This dissertation makes the following contributions:

- Identifying metrics and runtime characteristics that relate to the overhead associated with fine-grained communication in HPX.
- Designing a dynamic policy that makes task inlining decisions.
- Showing the impact of task inlining on different processor architectures.
- Providing a methodology to derive largely architecture independent constant $\lambda_{min}$ that allows controlling task granularity.
- Providing a methodology for extending Autochunking policy in HPX to determine the granularity of combined loops using $\lambda_{min}$ derived in the context of task inlining.

## 1.2. Publications

Parts of this dissertation contains previously published materials from IEEE and ACM that appeared in the following publications which have been incorporated throughout the dissertation. Permissions for reuse detailed in the Appendix.

- B. Wagle et al., "Methodology for Adaptive Active Message Coalescing in Task Based Runtime Systems," 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, 2018, pp. 1133-1140.
- B. Wagle et al., "Runtime Adaptive Task Inlining on Asynchronous Multitasking Runtime Systems," 48th International Conference on Parallel Processing (ICPP 2019), Kyoto, Japan, 2019.

## 1.3. Dissertation Outline

This rest of the dissertation is organized as follows: Chapter 2 presents additional background information along with an overview of HPX, the exemplar Asynchronous Many-Task runtime system used in this dissertation. Chapter 3 presents details pertaining to active message coalescing in HPX along with metrics and runtime characteristics related to remotely executed task. Chapter 4 presents design and implementation of dynamic policies for task inlining followed chapter 5 where an automatic parallel loop chunking policy is presented. A survey of related work is presented in chapter 6 and finally chapter 7 concludes the dissertation.

## Chapter 2. Background

In an ideal strong scaling scenario, a parallel application would run twice as fast by doubling the number of processors. However, Amdahl's law [9, 10], which states that scalability of an application is limited by the serial portion of the code, imposes a theoretical upper limit on the speedup a parallel application can achieve. Similarly, the concept of weak scaling is introduced by Gustafson's law [11] and states that as the problem size is increased, parallel work increases accordingly. In an ideal weak scaling scenario an application would be able to handle double the amount of work if the resources are doubled. However, ideal scaling behaviors are not seen in practice and deviation from ideal can be broadly attributed to the following factors as outlined in the ParalleX [6] model:

- **S**tarvation or the lack of concurrent work available in the system to keep all of the resources busy
- **L**atencies related to accessing services and resources
- **O**verheads of parallel execution which would not be present in sequential execution
- **W**aiting for contention resolution due to over-subscription of shared resources

All the above factors contribute to the compute resources being idle either due to lack of work, delays in accessing services or waiting for a shared resource to be available. The overheads of parallel execution where the runtime system performs work unrelated to the actual computation to achieve parallel execution also adds additional delays. HPX [7], an Asynchronous Many-Task runtime system used as an exemplar runtime system throughout the dissertation, is the first implementation of the ParalleX model and attempts to alleviate application scalability issues in order to extract maximum possible parallelism from the system. HPX exposes a concurrency and parallelism API that is consistent with the ISO C++ standard. HPX parallel applications can run on both a single machine as well as a cluster with hundreds of thousands of nodes. A detailed description of HPX and its implementation details can be found in the following publications [6, 7, 12]. A gentle overview of HPX useful to the comprehension of this dissertation is provided in the subsequent section.

## 2.1. HPX Runtime System

The design of HPX mainly revolves around using fine-grained tasks running on top of kernel threads via a lightweight scheduler that supports work stealing, applying local constraint based synchronization among tasks rather than global barriers, using active messages [13] for executing tasks wherever data is located and a mechanism for addressing any object globally. Fine-grained asynchronous tasks allows for better flexibility in keeping the underlying CPU busy while another task is waiting on a resource effectively hiding the latencies associated with memory access, network etc. The use of local constraint based synchronization instead of global barriers allows parts of the application where synchronization is not needed to avoid waiting. The constraint on synchronization is placed locally, for example based on data availability, which makes sure only those tasks that are waiting for some data to be available are suspended. Unlike traditional message-passing scenarios, using active messages in HPX allows tasks to be executed in the location of the data rather than moving data to where the task is located avoiding data movement. Furthermore, each object in HPX is assigned a Global Identifier(GID) that is maintained throughout the lifetime of the object even if it is moved between nodes in the system.

The modular structure of HPX is shown in figure 2.1. HPX consists of a $Thread\ Scheduler$ responsible for scheduling lightweight tasks, a $Performance\ Counter$ framework used for instrumentation purposes, a $Parcel\ Transport\ Layer$ for handling message passing and remote method invocations, lightweight $Local\ Control\ Objects$ ($LCOs$) for synchronization among tasks and an $Active\ Global\ Address\ Space$ ($AGAS$) for addressing object across nodes. The $Performance\ Counter$ framework is able to gather performance information from the whole system which can be used for the purpose of debugging, post-mortem analysis as well as for runtime adaptive purposes.

HPX exploits parallelism by executing lightweight tasks scheduled on top of the kernel threads. By default, HPX creates one kernel thread per core. The HPX scheduler schedules the lightweight tasks on top of these kernel threads. HPX tasks are non preemptive and

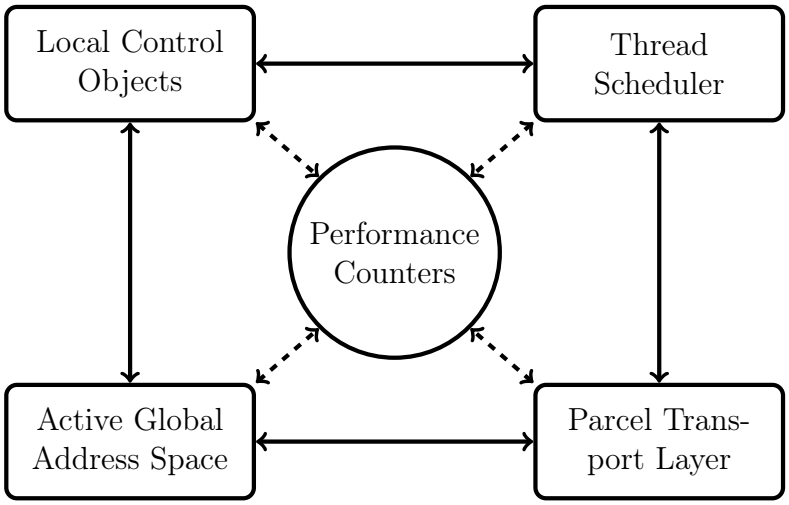



Figure 2.1. The modular structure of HPX runtime system

are stopped either when they run to completion or voluntarily yield their execution. An implication of the non-preemptive nature of HPX tasks is the fact that the tasks have to be short-lived or voluntarily yield occasionally to allow for fair scheduling. A task in HPX is also called *HPX-Thread* as it is a fully conferment implementation of the C++ standard thread, has its own stack and support calls to yield , suspend and resume [12].

Asynchrony in HPX is managed via *futures* [7, 14]. A future is a placeholder for the result of some computation that is not yet ready. A task requesting the result of a future is suspended if the result is unavailable. When the future becomes ready, wherein the results of the computation is available, the suspended tasks are resumed. Another important feature of HPX is the *dataflow* [15, 16] utility. HPX makes use of dataflow objects for managing data dependencies. A dataflow waits until a provided set of futures have become ready before executing a predefined callable which relies on the results referenced by the futures. Futures and dataflows are the prominent Local Control Objects in HPX among others such as mutexes, spin-locks, barriers and semaphores.

Remote task invocation in HPX is performed via parcels. A parcel is a form of an active message [13]. A parcel is created when a method, called *action* in HPX terminology, is called remotely. A parcel has four components: the *destination address* which is the location

where the method is to be executed, *action* which is the method to execute, *arguments* are the parameters of the method and *continuations* are optional objects that are executed after the main method in the parcel terminates. In order to transmit a parcel over the network, a parcel goes through a serialization process and is converted into a stream of bytes which is then transmitted using existing network protocols. At the receiving end, a de-serialization process reconstructs the parcel from the received sequence of bytes. The parcel is then converted into a task and placed in the scheduler queue for execution. The parcel layer is responsible for creating the parcels as well as converting a received parcel into a task.

HPX provides a system wide support for gathering performance information, known as the performance counter framework. This feature is used to extract information about the state of the application and runtime and is useful for instrumentation and debugging purposes. In addition, HPX and the performance counter framework integrate with APEX(*Autonomic Performance Environment for eXascale*) [17], which provides additional measurements and a policy engine that enables runtime adaptive capabilities. APEX is an external library which gathers performance information from the runtime system. This information can then be recorded for post-mortem analysis or used as inputs to the APEX policy engine. APEX uses an event based introspection API where an event is triggered either periodically or at a defined point in the application code. Users can define policies which respond to these events based on the current state of an application.

This work assesses overheads associated with fine-grained task execution in HPX and highlights methodologies to control the granularity of the tasks. In the subsequent section we will look at how the granularity of tasks effects the performance of a parallel application written in HPX.

### 2.2. Effects of Task Granularity in HPX

The granularity of the tasks can dictate the overall performance of an application. In this section, we will look at how the performance a HPX parallel application varies when the

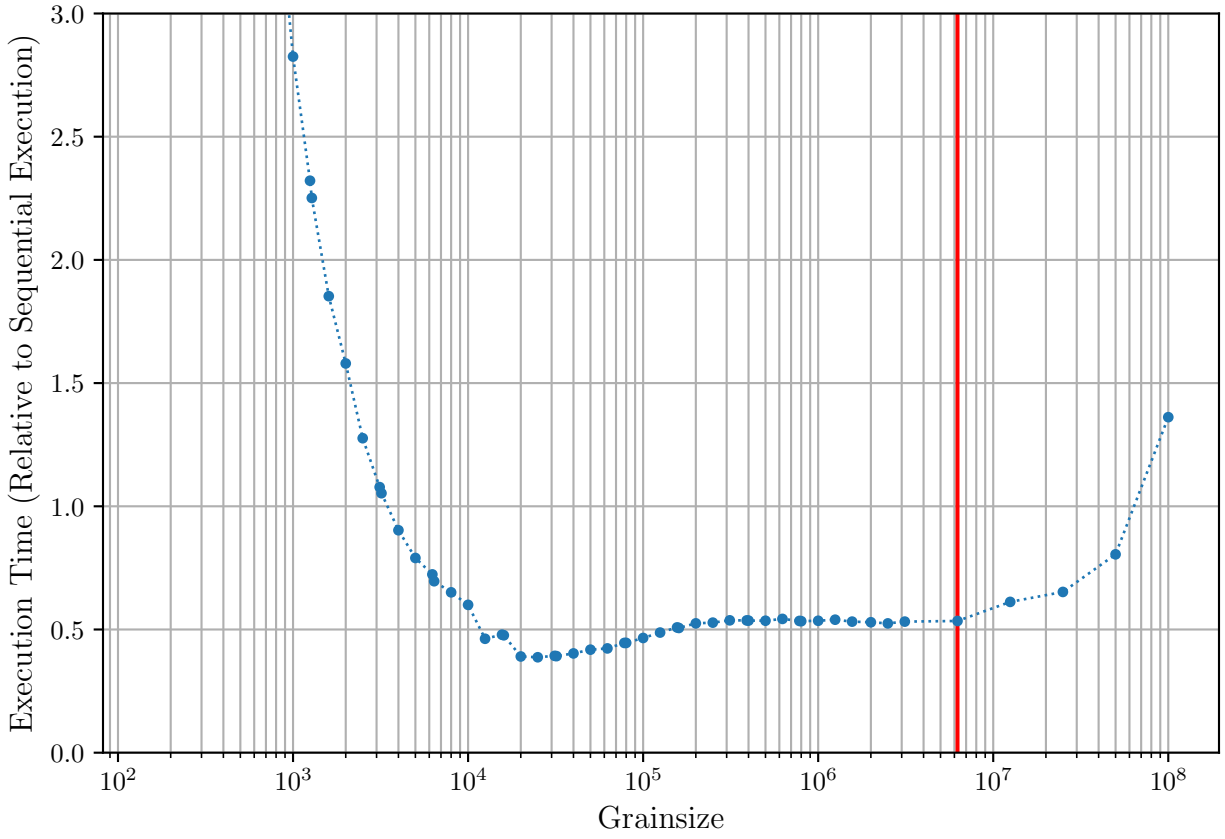



Figure 2.2. Execution time relative to sequential execution for the stencil application in HPX for various grainsize plotted using the blue line. All data points below 1.0 represent faster execution compared to sequential execution. The red vertical line indicates the grainsize that would be chosen if the total work was equally divided among the processing units.

granularity of the task is varied. For this demonstration, we use the one dimensional heat stencil[1] example in the HPX repository[2]. In the stencil example, the data points were partitioned such that one HPX task is created for each partition. The grainsize or the amount of work performed by each HPX task in the example can be controlled by varying the data points per partition.

Figure 2.2 shows the execution time relative to sequential execution time for the stencil example running on 16 cores with a total of 100000000 datapoints. The grainsize in figure 2.2 is controlled by controlling the data points per partition. For example, a grainsize of 1000 represents 1000 data points per partition. Larger grainsizes indicate fewer tasks

[1]https://github.com/STEllAR-GROUP/hpx/tree/master/examples/1d_stencil/1d_stencil
[2]https://github.com/STEllAR-GROUP/hpx

of longer duration were executed whereas smaller grainsize indicate more tasks of shorter duration were executed.

It is seen from figure 2.2 that increasing the granularity of the task improves application performance up until a certain point after which the improvement flattens. As the granularity is further increased, the performance degrades. The portion of the graph towards the left is dominated by overheads associated with parallel execution as the work contained in the task is not able to amortize the cost of overheads. The portion of the graph towards the right is dominated by starvation where not enough parallel work is available in the system to keep all the processing units busy. It is also seen from figure 2.2 that the region between the points where the cost of overheads is amortized and starvation kicks in, better performance is seen with lower grainsize. The red vertical line in the figure indicates the grainsize that would be chosen if the total work was equally divided among the processing units as is done traditionally. However, this may not be the optimal grainsize as seen from the stencil example. Increasing the granularity beyond a limit can result in degradation of performance. In chapter 4 of the dissertation, with regards to locally executed tasks, we will look at methodologies for estimating minimum granularity of HPX tasks in the context of task inlining.

# Chapter 3. Active Message Coalescing

In this chapter, we explore overheads associated with tasks that are executed on a node other than the one where it was created. As we move towards *exascale* computing, where tens of thousands of nodes will work together in solving complex scientific problems, asynchronous many-task runtime systems have carved out its own space alongside the de-facto standard of High Performance computing, MPI [3]. The success of Asynchronous Many-Task runtime systems is based on the fact that most algorithms can be decomposed into fine grained units of work that can be executed by the runtime system. A side effect of creating fine grained units of work in a large scale distributed application is fine-grained inefficient communication patterns. If we are sending a large number of messages in quick succession, the overheads associated with fine-grained tasks rapidly aggregates. In the context of Asynchronous Many-Task runtime system, where fine grained communication is ubiquitous, reduction of overheads introduced by the transmission of information is vital. Any improvements that can be made in this context have the potential to improve the overall execution time of the distributed application.

Coalescing messages allows users to combine small messages into large ones that effectively send the same amount of data but keep the per message overheads at a minimum. Although programmers can manually coalesce messages to optimize their applications, the effort required to correctly achieve this is quite high and is practical only in small and simple applications. Recent work such as Active Pebbles [18] , AM++ [19] and Charm++ [20], have implemented some form of message coalescing solutions provided by runtime systems. Such solutions are largely beneficial in terms of reducing program complexity and coding time. A programmer would simply enable message coalescing and the runtime would intelligently coalesce messages bound to the same destination.

Parts of this chapter were previously published as B. Wagle, S. Kellar, A. Serio and H. Kaiser, "Methodology for Adaptive Active Message Coalescing in Task Based Runtime Systems," 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, 2018, pp. 1133-1140.

In the rest of this chapter, we look at the implementation of message coalescing in HPX. Furthermore,we look at methodology for estimating the overheads associated with fine-grained communication in HPX. We devise metrics that relate to the overheads associated with active messages in HPX. We use message coalescing as a technique to reduce active message overheads in HPX and show that the metrics derived in this chapter are useful to estimate the cost of overheads associated with sending and receiving large number of active messages in HPX. The metrics and techniques defined in this chapter have the potential to be used as a basis for the adaptive tuning of a broad set of messaging parameters. In the following section, we look at the implementation details regarding active message coalescing in HPX.

### 3.1. Parcel Coalescing in HPX

A parcel [21] is a form of active message in HPX as described in section 2.1. Individual parcels are grouped together to form a larger coalesced message which is reconstructed into the original entities at the receiving end. Figure 3.1 shows a diagrammatic representation of parcel coalescing in HPX. Here, individual parcels are grouped together to form a larger message containing the coalesced parcels. The coalesced message is then serialized and sent to the destination. At the destination, individual parcels are reconstructed and placed in the HPX scheduler queue. Implementation details and further information regarding parcels and serialization in HPX can be found in [12].

One caveat of parcel coalescing is determining how many parcels to coalesce in a single send. A coalesced parcel can be defined by either the size of the buffer, number of parcels, a timeout or any combination of these criteria. The design of parcel coalescing in HPX revolves around two parameters. First, the length of the parcel queue and second, the wait time. The length of the parcel queue outlines a suggested number of parcels to be coalesced before being sent. The wait time dictates the number of *microseconds* to wait for the queue to be full before sending the current queue of parcels as one message. Hence, coalesced parcels are sent either when the parcel queue is full or when the wait time expires.

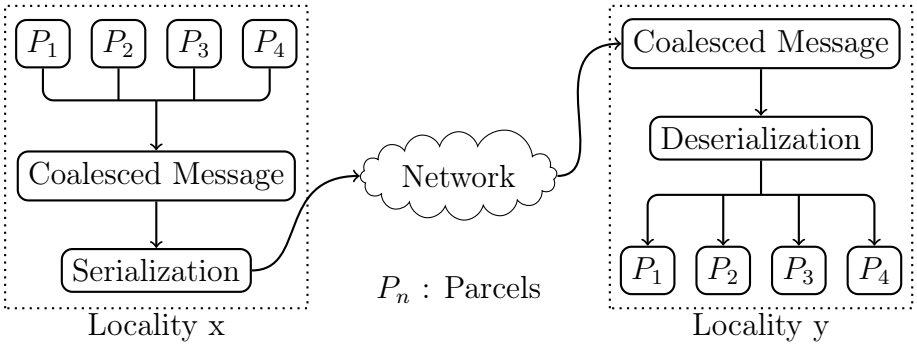



Figure 3.1. Diagrammatic representation of parcel coalescing

Additionally, a limit on the maximum size of the buffer is applied in order to avoid memory overflow errors. The algorithm for parcel coalescing in HPX is shown in algorithm 1.

The accuracy of the *wait timer* responsible for signaling the send operation for the parcels waiting in the queue is an important factor in the overall design of the parcel coalescing module. The *wait timer* is designed using the *deadline_timer* from the Boost[1] library that allows the timing mechanism to run in its own dedicated hardware thread. It is not desirable to implement the *wait timer* as a HPX task because HPX tasks are not preemptive as discussed in section 2.1. This means that HPX tasks can only be stopped at completion or through voluntary yielding. In a scenario where the HPX scheduler is busy due to large number of tasks, the task responsible for *wait timer* may not be scheduled immediately resulting in lower accuracy of the *wait timer*. In order to verify the accuracy of the *wait timer*, an experiment was performed where a timer was created and set to expire after certain interval. It was observed from the data obtained from the experiment that the *wait timer* fires within on average of 33µs of the desired fire time. This guarantees that on an average, the *wait timer* in the HPX parcel coalescing plugin expires within 33µs of the desired wait time.

Another important design consideration when implementing parcel coalescing is when to disable it. The communication pattern of a real life application may change throughout its lifetime. The application may generate large number of parcels at certain points,

[1]https://www.boost.org/

**Algorithm 1** Parcel coalescing policy

```
num_parcels                                  ▷ number of parcels to coalesce in a message
interval                                     ▷ wait time in microseconds
parcel_state                                 ▷ state of arriving parcel
time_last_parcel                             ▷ time since last parcel
if time_last_parcel > interval then
    SEND_PARCEL()
end if
switch parcel_state do
    case First:
        TIMER.START(interval)
        QUEUE_PARCEL()
    case !First||Last:
        QUEUE_PARCEL()
    case Last:                               ▷ queue is full
        TIMER.STOP()
        SEND_PARCEL()                        ▷ send queued parcels as one
```

whereas, there may be periods in the application where the number of parcels generated is small. In the design of the parcel coalescing module in HPX, coalescing is performed only if the time between parcel generation is less than the wait time. This feature effectively disables parcel coalescing in cases where parcel generation is sparse. It is important to disable parcel coalescing in cases where parcel generation is sparse because the performance will be negatively impacted when the application has to wait for the parcel queue to be flushed by the *wait timer*. Furthermore, since parcel coalescing is beneficial in the specific case where large number of parcels are generated, parcel coalescing is implemented in the form of a plug-in rather than incorporating it into the core of HPX. This keeps HPX flexible by only enabling parcel coalescing plug-in when needed. Also, parcel coalescing has been implemented on per *action* basis and is effective only if explicitly enabled for a particular HPX *action*. Parcel coalescing for a particular *action* can be enabled with minimal change to the existing code by adding the macro *HPX_ACTION_USES_MESSAGE_COALESCING()* as seen in line 8 in listing 3.1.

During the course of this study, the following performance counters specific to parcel coalescing were incorporated into HPX:

- */coalescing/count/parcels* that return the number of parcels associated with a particular *action*,
- */coalescing/count/messages* that return the number of messages generated for a particular *action*,
- */coalescing/count/average-parcels-per-message* that return the average number of parcels sent in a message for a particular *action*,
- */coalescing/time/average-parcel-arrival* that return the average time between arriving parcels for a particular *action*,
- */coalescing/time/parcel-arrival-histogram* that return a histogram representing the gap between parcel arrival for a particular *action*.

Performance counters specific to coalescing provide intrinsic information about the application that can be used for debugging and optimization purpose. The performance counters listed above were used for preliminary analysis of parcel coalescing. The above counters also aided in debugging our implementation of parcel coalescing.

### 3.2. Network Performance Metrics

This work develops metrics for measuring network overhead of an application. In the context of this work, overhead is defined as the time spent processing information to be communicated across the network. This processing time we call *background work*. While informative, the time spent processing *background work* is insufficient to gauge the effects network overhead on the application. An increase in time spent on *background work* may only indicate a change in application state, eg. communication phase of an application. To understand the influence of overhead, we must look at the ratio of background work to overall execution time. The background work time paired with the overall execution time of the application determines the actual influence of network overheads. The proportion of time spent on overheads to the overall runtime indicates whether significant improvements are possible via a reduction of network overheads.

Parcel coalescing is useful as it reduces the overhead cost per message. In an applica-

tion that sends millions of messages during its execution, this reduction will be extremely beneficial. After implementing parcel coalescing, we analyzed its effect on the overhead associated with sending and receiving messages. We used two applications, Parquet [22] and a toy application. Details about these applications are provided in section 3.3. Using the Performance Counter Framework provided by HPX, we obtained intrinsic information about the applications in real time. This section details the metrics we gathered to evaluate the network overheads.

### 3.2.1. Execution Time

We first measured the execution time of our test applications while varying the number of parcels to coalesce in a single message and the interval to wait before flushing the queued parcels. The size of the problem in each run was kept constant, hence the same number of parcels were generated in each run. The difference between runs was simply the number of messages sent as determined by the coalescing parameters.

### 3.2.2. Task Duration

Next we looked at the overall time spent on executing each HPX-thread or tasks including the overhead. We define task duration using the following equation:

$$t_d = \sum t_{func} \tag{3.1}$$

where $\sum t_{func}$ is the total time spent by the HPX scheduler executing each HPX-thread.

### 3.2.3. Task Overhead

We then looked at the average time spent on thread management for each HPX-thread or tasks. All communication in HPX is done via tasks. Task overhead [23], is obtained from the */threads/time/average-overhead* performance counter. We calculate task overhead using the following equation:

$$t_o = \frac{\sum t_{func} - \sum t_{exec}}{n_t} \tag{3.2}$$

where $\sum t_{exec}$ is the time spend by the HPX scheduler doing useful work and $\sum t_{func}$ is the task duration as defined in equation 3.1 and $n_t$ is the number of executed HPX threads. We observed a positive correlation between task overhead and overall execution time of our test applications for various coalescing parameters.

### 3.2.4. Background Work Duration

After establishing that task overhead has a positive correlation with the overall execution time, we separated the network related overhead from other overheads. HPX performs network related tasks such as packaging a parcel into a message, serialization, handshaking and locality resolution in the form of background work. We define total time spent doing background work as the background work duration and it is obtained using the following equation:

$$t_{bd} = \sum t_{background-work} \tag{3.3}$$

Background work duration can be queried using the performance counter */threads/background-work* and was added to HPX as a part of this study.

### 3.2.5. Network Overhead

The network overhead, obtained from the performance counter */threads/background-overhead*, is the ratio of thread background work duration to task duration. HPX measures $\sum t_{\text{background-work}}$, the running sum of time spent on performing network related duties of each HPX-thread, and $\sum t_{\text{func}}$, the running sum of total time to complete each HPX-thread. Network Overhead is shown in equation 3.4.

$$n_{oh} = \frac{\sum t_{background-work}}{\sum t_{func}} \tag{3.4}$$

Here, $\sum t_{background-work}$ is the total time spent performing network related work and $\sum t_{func}$ is the total time to reach the completion of each HPX-thread. The network overhead performance counter, */threads/background-overhead* was added to HPX as a part of this

Table 3.1. Marvin node specifications

| Marvin Thin Compute Node | |
|---|---|
| Microarchitecture | SandyBridge |
| CPU | Xeon E5-2450 |
| Total Number of CPUs | 2 |
| Total Number of Cores | 16 |
| Frequency | 2.1 GHz |
| Memory | 48 GB |

study.

In subsequent sections, we use the Network Overhead metric defined in equation 3.4 in order to measure network overhead of our test applications. Parcel Coalescing is used to demonstrate that the reduction of network overhead increases overall application performance. The control parameters of parcel coalescing can be modified which, in turn, results in a corresponding increase or decrease of network overhead. For network intensive applications, changing network overhead has a demonstrable effect on the application's execution time. Parcel coalescing attempts to minimize network overhead by sending larger messages across the network via aggregation of smaller parcels into one large message. In the instance of a high volume of parcels in a short window of time this can significantly reduce network overheads.

## 3.3. Experimental Results

We analyzed the effect of overheads associated with sending and receiving messages using two applications, a toy application and a real life scientific application Parquet [22]. Using the Performance Counter created to represent the metrics defined in the previous section, we obtained intrinsic information about the applications in real time. For our evaluation, we used Marvin thin compute nodes of the ROSTAM [24] cluster located at Louisiana State University. The hardware specifications for Marvin is listed in table 3.1.

### 3.3.1. Toy Application

In order to test the effectiveness of parcel coalescing on HPX and its effect on the Network Overhead metric defined in equation 3.4, we used a toy application that sends millions

```
//Create Action
complex<double> get_cplx()
{
  return complex<double>(13.3,-23.8);
}

HPX_PLAIN_ACTION(get_cplx,actn);
HPX_ACTION_USES_MESSAGE_COALESCING(actn);

//Create instance of the actions
actn act;

vector<hpx::future<complex<double>>> vec;
vec.reserve(numparcels);

//Find the other locality
auto localities=hpx::find_remote_localities();
auto other=localities[0];

int num_repeats=4;
//Repeat num_repeats times
for (int j = 0; j < num_repeats; j++)
{
  // Start of a phase
  for (int i = 0; i < numparcels; ++i)
  {
    vec.push_back(hpx::async(act, other));
  }
  //Wait for all the tasks to complete
  hpx::wait_all(vec);
  // End of a phase
}
```

Listing 3.1. Artificial example application used to generate and send parcels from one node to another.

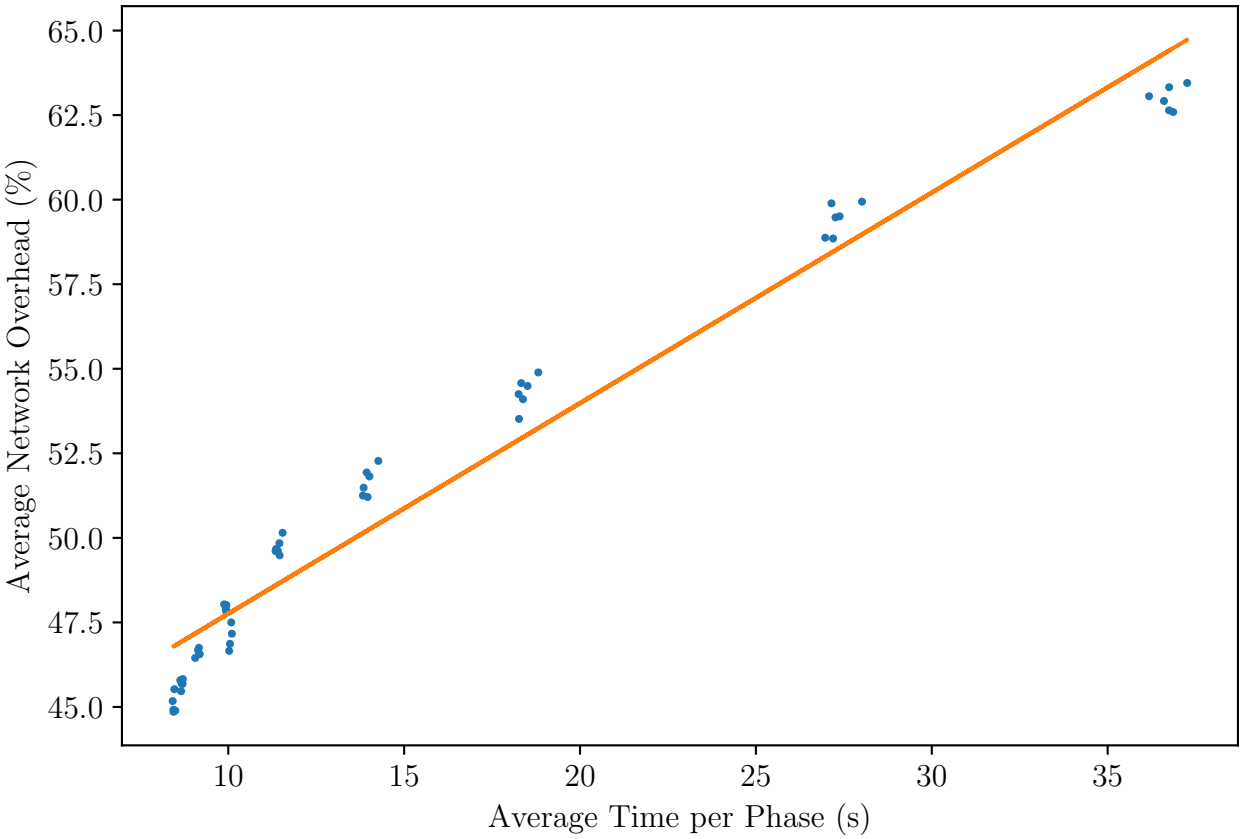



Figure 3.2. Scatter plot of the average network overhead per phase vs average execution time per phase for the toy application. Each dot represents a set of parcel coalescing parameters. Average overhead is the average for four phases. As the network overhead decreases, the execution time also decreases. A Pearson's correlation coefficient of 0.97 indicates a strong positive correlation between network overhead and runtime.

of messages containing a single complex double with no direct dependencies between the messages. This example simulates an application where the network overhead is high and is an ideal candidate for testing the effectiveness of parcel coalescing. A condensed version of the code for the toy application is in listing 3.1. It shows two nodes sending a million messages to each other and this process is repeated four times. We define the process of sending a million message as a phase as indicated by line 24 to line 31 in listing 3.1. Hence, the toy application sends a million message in a phase and there are four phases. Throughout the lifetime of the toy application, four million messages are sent and received by a node.

We measured the network overhead at specific phases for the toy application using

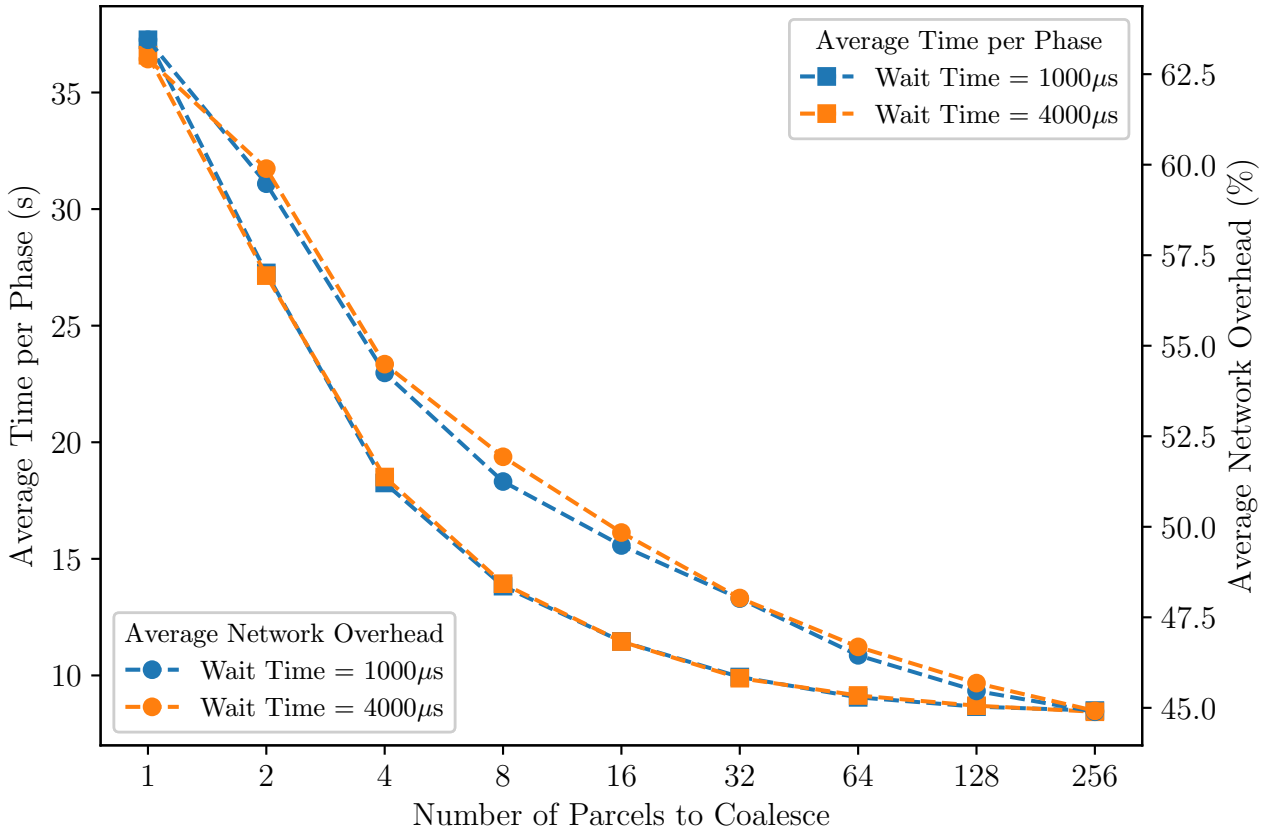



Figure 3.3. Changes in average time per phase along with the average network overhead for the toy application with various values of number of parcels to coalesce in a single send and wait time of 1000µs and 4000µs. The average network overhead follows the same trend as the average time per phase.

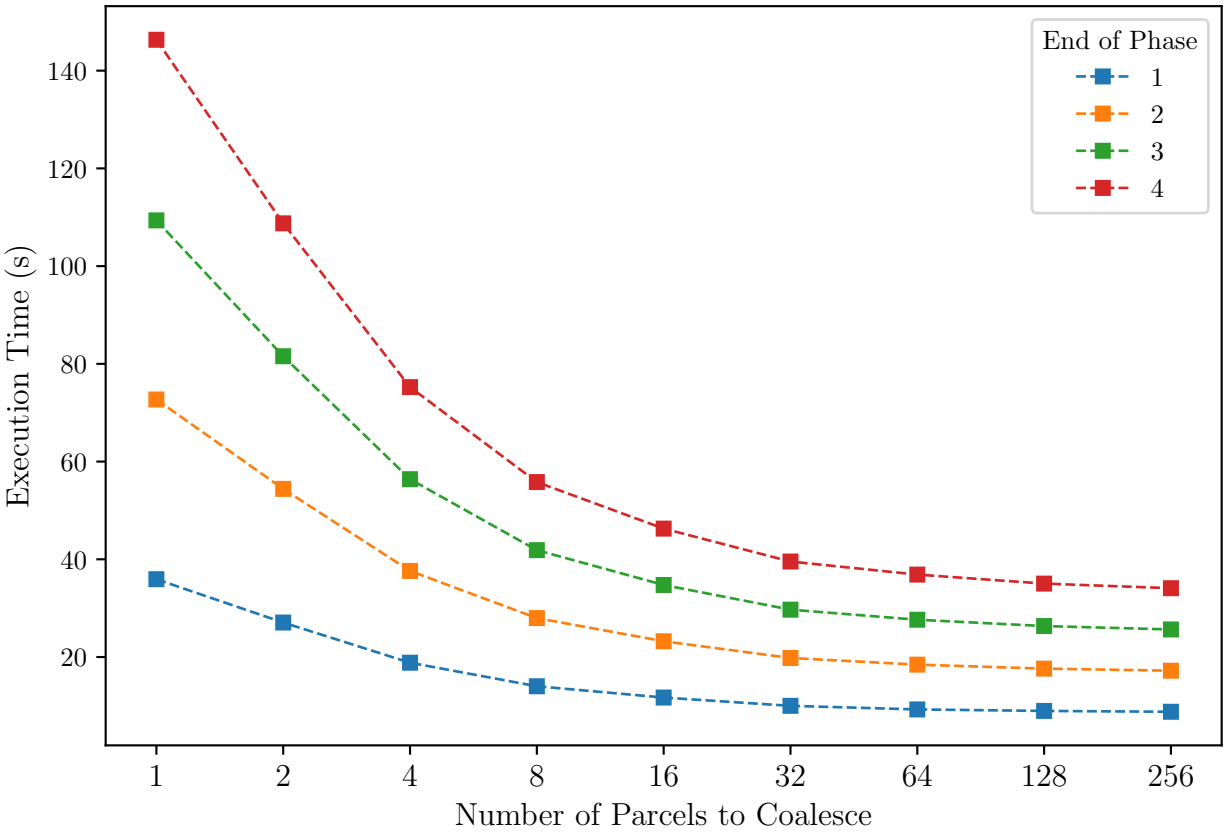



Figure 3.4. Time to reach the completion of a particular phase in the toy application for various values of number of parcels to coalesce in a single message with a wait time of 4000µs. In this example, as more parcels are coalesced, the time to reach the completion of a phase decreases.

equation 3.4. Figure 3.2 shows the scatter plot of average network overhead per phase vs the average execution time for all sets of parcel coalescing parameters explored in this work. In the experiments, the number of parcels to coalesce in a single send was set at 1, 2, 4, 8, 16, 32, 64, 128 and 256. Similarly, for each value of number of parcels to coalesce, the wait time was set at 1µs, 1000µs, 2000µs, 3000µs, 4000µs and 5000µs. It is seen that the set of parcel coalescing parameters that result in lower execution time also has lower value for network overhead and the parameters resulting in higher execution time has higher value for network overhead. The Pearson's correlation coefficient for our data set was 0.97 which indicates that network overhead and execution time are strongly correlated. We can confidently conclude that larger reported network overhead results in longer execution times. Figure 3.3 also highlights the same fact.

It is also desirable to see the relationship between parcel coalescing parameters and application runtime. Figure 3.4 shows the execution time for various values of number of parcels to coalesce in a single messages. The fastest execution time occurs with the largest values of number of parcels to coalesce. This result reflects the toy application's lack of dependency with any other computation or communication. This is further highlighted by our observation that changing the wait time has negligible effect on the execution time. This is due to the fact that the toy application generates the parcels in quick succession such that the parcel queue is almost always filled and the wait time rarely expires. Next we look at a real life scientific application that has phases of computation along with communication. Furthermore, unlike the toy application, there are dependencies between the computation and communication which should result in a different set of optimal coalescing parameter compared to the toy application.

### 3.3.2. Parquet Application

The second application used in the evaluation was the Parquet [22] application. The self consistent parquet method is a complex physics simulation. The goal is to identify parameters which control the emergence of interesting quantum phenomena in strongly correlated materials. The simulation requires the use of many rank-3 tensors composed of complex doubles. The linear dimension ($N_c$) of the simulation controls the tensor size. The tensor contains $N_c^3$ complex doubles. The memory required by the simulation can approach terabytes. Accommodating such large quantities of data requires the simulation to be executed on multiple nodes. Throughout the simulation, all the data from each node must be broadcast to the other nodes. The rotation phase sends $8 * N_c^2$ parcels containing $N_c$ elements. No message depends on another and they can be sent in parallel.

For the trial simulation executed on four nodes, $N_c = 512$ was chosen as it uses a non trivial amount of memory and it exposes high network utilization. Tests indicate the timing of an individual run will not be likely to vary much from the averages reported. In order to test the precision of the application runtime, we coalesced 4 parcels into a single

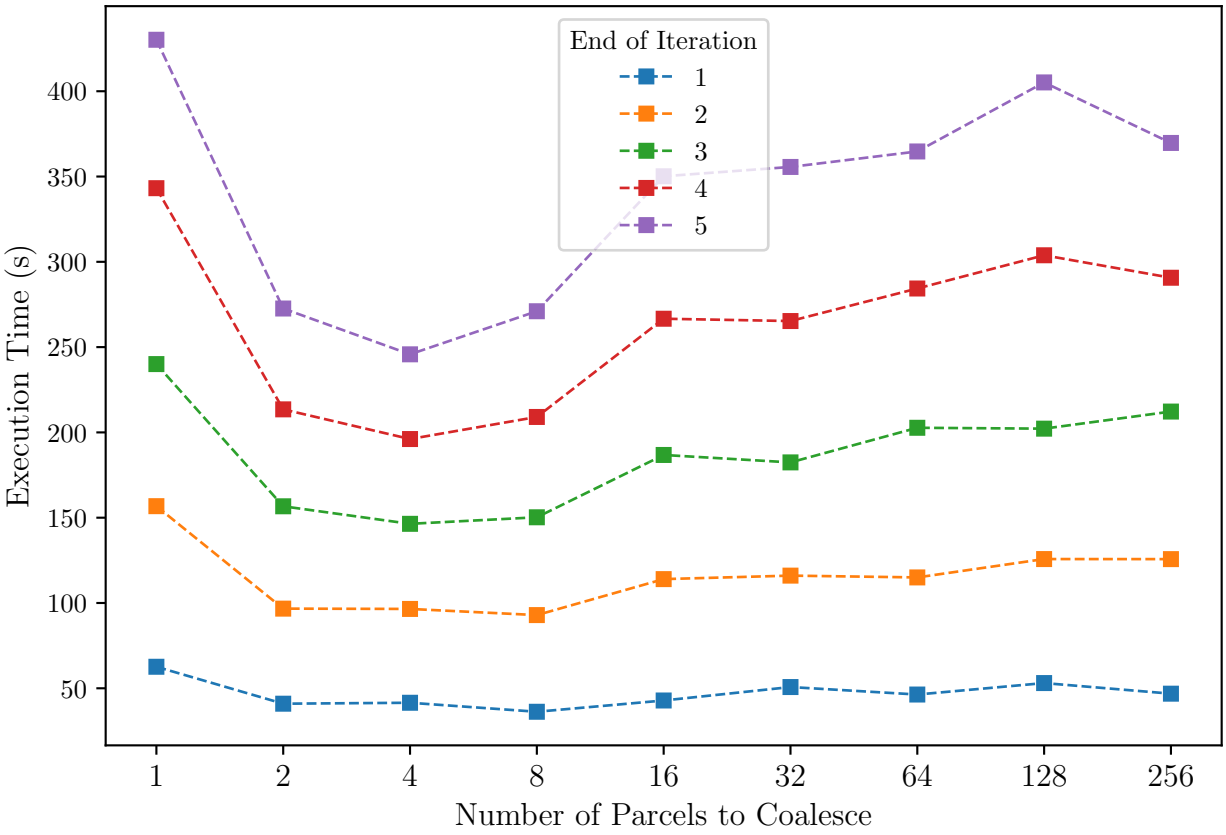



Figure 3.5. Time to reach the completion of different iterations in the parquet application for various numbers of parcels coalesced in a single message with a wait time of 4000µs. Each color indicates a different iteration. There is a clear decrease in overall runtime from coalescing one parcel in a message to coalescing two. The minimum runtime is found when coalescing four parcels in a message after which the runtime increases due to a sub-optimal choice of parcel coalescing parameters.

message and waited 5000µs before flushing the parcel queue. We ran the experiment 100 times. The calculated Relative Standard Deviation of the trial was less than five percent which indicates that the random fluctuations of an individual run should not influence the trends reported. Measurements of network overhead and total execution time demonstrate the importance of parcel coalescing in a communication heavy problem. To account for the random nature of any application that involves heavy network traffic, the application was run three times for each set of parameters. The following results show the averages of the measured values from the three independent runs.

Figure 3.5 shows the variation in overall time to complete different iterations of the parquet application coalescing different numbers of parcels in a single message with a wait

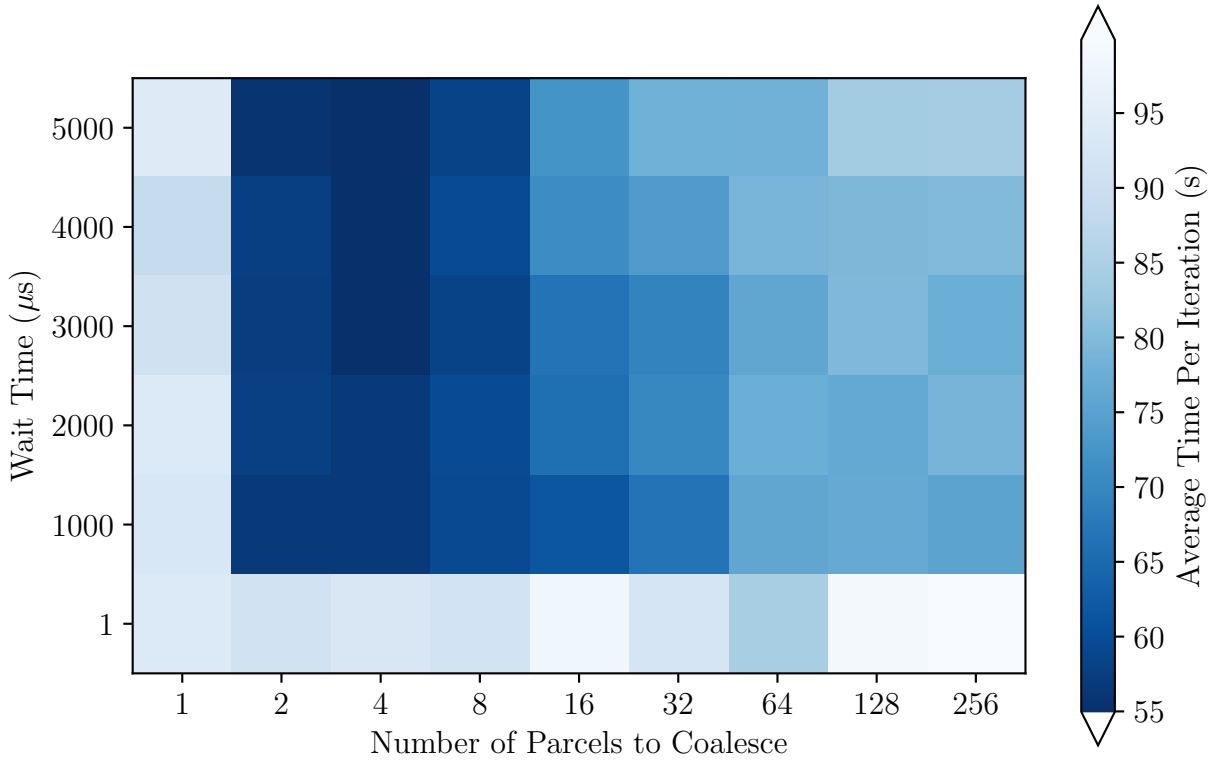



Figure 3.6. Average time per iteration for different numbers of parcels to coalesce into a single message and increasing wait times before flushing the parcel queue. Parcel coalescing is effectively disabled when a message contains only a single parcel or only 1 μs wait time before flushing the parcel queue. This produces slower execution times as seen in the bars along the horizontal and vertical axes.

time of 4000μs. We observed a clear decrease in runtime by coalescing two parcels in a message. Increasing the number of parcels to coalesce improved the runtime further. It was observed that coalescing four parcels in a message resulted in the minimum time. Further increasing the number of parcels in a message adversely affects the runtime. These trends are more pronounced in the later iterations due to cumulative effects. In order to further understand the relationship between parcel coalescing parameters and the overall runtime of the parquet application, we performed a parameter sweep running the parquet application with increasing the value of the parcel coalescing parameters until the execution time showed a clearly increasing trend. As seen in figure 3.6, bands along the axes where number of parcels to coalesce in a single message is one or when wait time is set at 1μs highlight the largest runtimes. This choice of parameters effectively disable parcel coalescing thus

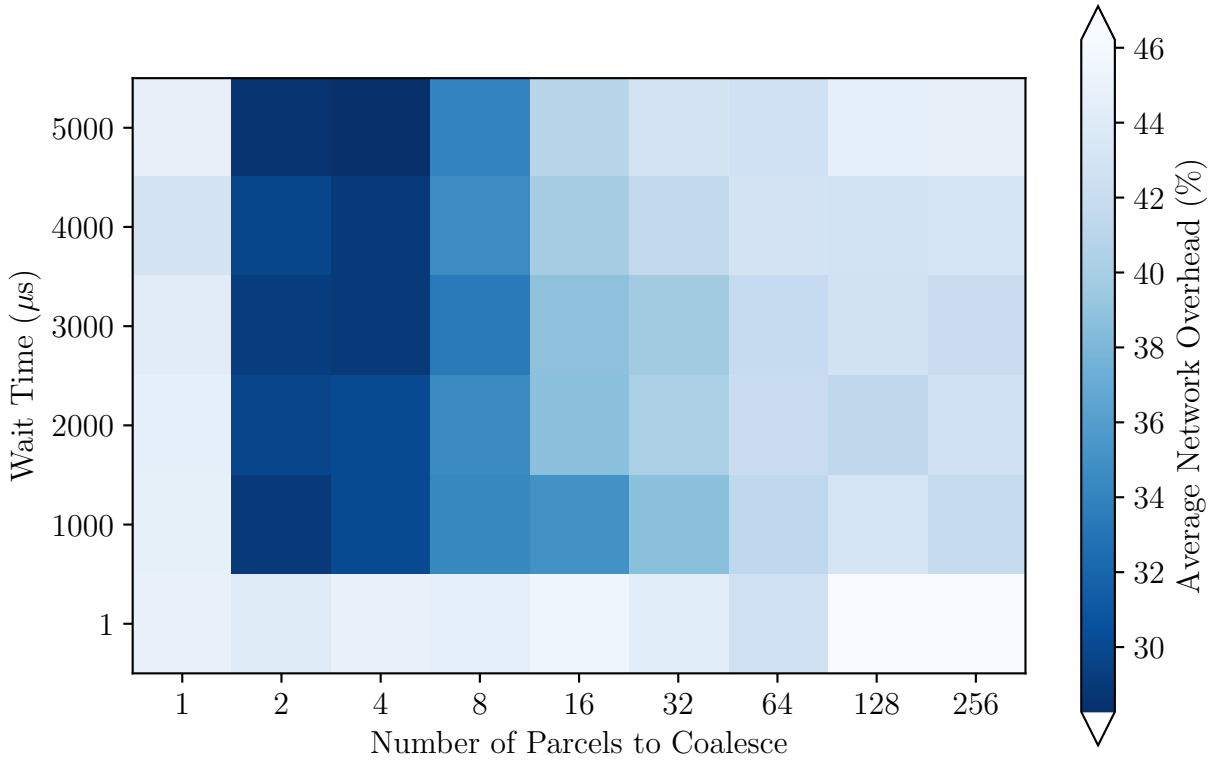



Figure 3.7. Average network overhead per iteration for different numbers of parcels to coalesce into a single message and increasing wait times before flushing the parcel queue. Parcel coalescing is effectively disabled when a message contains only a single parcel or only 1 µs wait time before flushing the parcel queue. This produces highest overheads as seen in the bars along the horizontal and vertical axes.

resulting in the large runtime seen. We get an immediate reduction in runtime with two parcels to coalesce in a single message. The maximum reduction in runtime was seen with coalescing four parcels in a message and wait time of 5000µs. For the same runs, performance counters reported the network overhead as defined in equation 3.4. As seen in figure 3.7, bands along the axes where number of parcels to coalesce in a single message is one or when wait time is set at 1µs highlight the largest overheads as see in the bars along the vertical and horizontal axes. Parcel coalescing is effectively disabled when a message contains only a single parcel or only 1 µs wait time before flushing the parcel queue. Figure 3.8 graphs various values of coalescing parameters against the value of the average network overhead counter. It was seen that the parcel coalescing parameter that resulted in lower overhead additionally had lower execution time. Our calculated Pearson's correlation

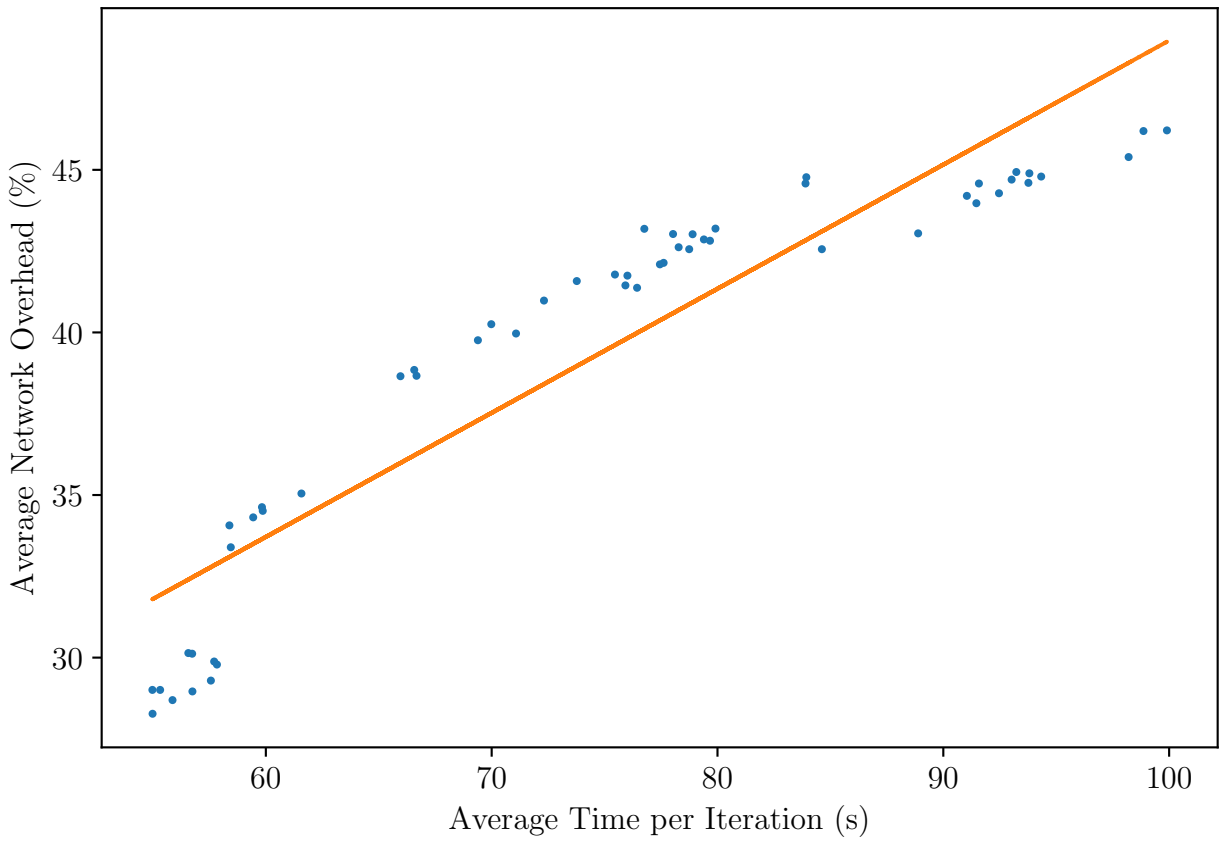



Figure 3.8. Scatter plot of average network overhead vs average time per iteration for the Parquet application. Each dot represents a set of parcel coalescing parameters. A Pearson's correlation coefficient of 0.92 was calculated indicating a strong positive correlation between network overhead and runtime for the parquet application.

coefficient of 0.92 indicated a strong positive correlation. Most of the parameter sweeps result in larger overheads than the optimum parameter. This implies that an arbitrary choice of parcel coalescing parameters will likely result in sub-optimal performance. The choice of parcel coalescing parameters must therefore be done carefully.

### 3.4. Summary

The performance of applications that generate large numbers of small parcels in task based runtime systems such as HPX can be improved by coalescing these small parcels into larger ones. This improvement is largely due to reduction in network overheads as fewer messages are created. We implemented parcel coalescing in HPX as a means to reduce the cost of overhead associated with sending and receiving messages. Our implementation of parcel

coalescing in HPX provided marked reduction in total runtime for the toy application as well as a real physics simulation, Parquet. This work also presented methods to measure the network overhead within task based runtime systems. We were able to establish strong positive correlation between the network overhead measured using our metric and the overall runtime of both applications. We showed that the benefits from parcel coalescing are due to the reduction in overheads associated with message transmission.

We demonstrated a static approach towards message coalescing where the parameters were selected before the start of the application. However, such static approaches towards coalescing parameter selection can only provide limited gains in performance. Adaptive techniques are needed to make further reductions in application execution times. TRAM [25] has shown the effectiveness of automatic configuration parameter selection using PICS: A Performance-Analysis-Based Introspective Control System [26]. Their approach tested a set of configuration parameters for each iteration of the application and chose new parameters based on the performance measured during that iteration. This approach to adaptive tuning is only suited for iterative applications, and therefore, this technique is unable to consider the phase of the application.

The methodology introduced in this research improves upon the state of the art by introducing new intrinsic performance counters which provide the current state of the application in real time by querying the associated performance counters. Using information obtained from such counters, one can make a distinction between different communication phases of the application and select configuration parameters accordingly. Metrics identified in this research have shown a strong correlation with execution time of the test applications and can aid in evaluating network efficiency by giving us an intrinsic view of the underlying network overhead which would be difficult to measure using conventional methods. Our research gives the user an ability to assess performance from a perspective other than that of execution time. This allows a user to analyze an application in real time and observe the effect of varying parcel coalescing parameters on network overheads

at runtime. In the future, metrics and techniques defined in this research could be used as a basis for the adaptive tuning of a broad set of messaging parameters.

# Chapter 4. Task Inlining

In order to effectively parallelize an application, a deeper understanding of how parallelism is achieved is useful. In the context of HPX, our exemplar runtime system, parallelism is achieved by executing tasks of various granularity on top of operating system threads via a lightweight scheduler. We can think of the total execution time of a HPX parallel application as the work performed by these HPX tasks plus the cost of parallelization. We further think of the cost of parallelization as the cost of creating and managing these tasks that would not have been necessary in the case of sequential execution. In other words, the act of running an application in parallel introduces overhead costs and minimization of these costs are essential.

Task inlining [27] is one of the techniques that can be utilized in order to reduce overheads of task creation and scheduling. In this technique, a parent task completes the work designated for a child task in addition to its own. In doing so, the child task is never scheduled as a separate thread which circumvents the overheads associated with creating and managing the child task. As a direct result, inlining naturally increases the granularity of the parent task. The success of task inlining solely depends on the decision when to inline a task. If an aggressive task inlining mechanism is applied, an application may lose the available parallelism that directly contradicts the objectives of an asynchronous many-task runtime. On the other hand, if task inlining is done rarely then the application will face unnecessary task creation overheads. Therefore, the decision of when to inline a task can carry severe performance implications.

In this chapter, the effects of task inlining is studied in the context of HPX. HPX is capable of scheduling a new task asynchronously or synchronously in the parent task, which we will refer to as inlined execution. We will further look at methodology for utilizing per

Parts of this chapter were previously published as B. Wagle, M. A. H. Monil, K. Huck, A. D. Malony, A. Serio, and H. Kaiser, "Runtime Adaptive Task Inlining on Asynchronous Multitasking Runtime Systems," 48th International Conference on Parallel Processing (ICPP 2019), Kyoto, Japan, 2019. 

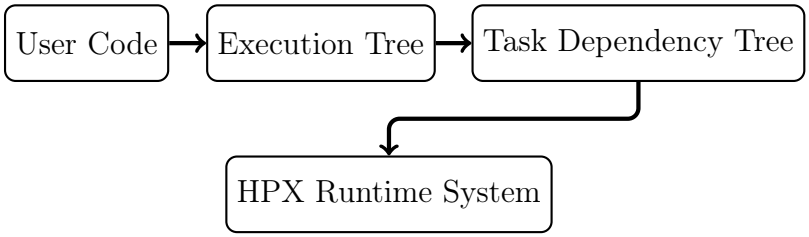



Figure 4.1. The workflow of Phylanx

task overhead for determining appropriate granularity of the parent task when inlining is performed. Phylanx, an array processing toolkit written on top of HPX, is used to experiment with our developed methods and is described in section 4.1 below. The main reason for using Phylanx for experimentation is that Phylanx has the notion of *primitives*, which are independent operations that work on provided data. By default, each of these primitives is scheduled as a new task, thus providing our experiments with a well-defined set of tasks of varying lengths.

### 4.1. Task Inlining in Phylanx

Phylanx [28,29] is a task based, asynchronous array computing toolkit designed to support machine learning applications. User code, written in Python, is transformed into a tree of Phylanx primitives known as an execution tree. A primitive is an object that can take an input, such as the result of a previously executed primitive, and exposes a method named *eval* that performs an operation on the object's inputs. Instead of returning the value computed by the primitive, however, the *eval* method will return a HPX future to the computed value. An execution tree, which is a collection of these primitives, describe the dependencies between all the operations in an application. In this formulation, the nodes of the execution tree are the primitives while the edges of the tree represent dependencies between them. The workflow of Phylanx is shown in figure 4.1.

During execution, Phylanx starts to evaluate the execution tree by calling the *eval* function on the root node. Each dependency of this primitive calls the *eval* function on each of its dependencies. This operation traverses the tree until a leaf node, or a node with no dependencies, is reached. It is important to note that as the execution tree is

being traversed the actual execution of the tasks have not yet begun. Rather a task graph of HPX futures is being created where each HPX future represents a dependency on a previous operation. Once the leaf nodes have been reached, the task graph then begins to execute, as the execution of a leaf primitive does not depend on the results of another calculation. The task graph is then summarily executed as the results of dependencies are met, eventually returning the result of the entire tree. As the evaluation of a child node is completed, the result of its execution is passed to the parent node. The result of the entire tree is ready after the root node has finished execution.

The design of Phylanx allows for a well-defined injection point for studying the impact of task inlining. As discussed earlier, Phylanx, has the notion of primitives, which are independent operations that work on provided data. Each primitive in Phylanx has a method called *eval* that performs the work contained in that primitive. The *eval* method of Phylanx uses HPX dataflow in order to launch a primitive's operations. HPX can decide whether to execute the *eval* method asynchronously as a new task or synchronously by inlining the work in the parent task. This allows for a runtime injection point where we can decide whether to execute a primitive's children asynchronously in a new task or synchronously by inlining the execution. Furthermore, Phylanx has been designed to support machine learning applications that are often iterative in nature. An iterative application implies that the same Phylanx primitives will be executed repeatedly. Phylanx also has built in performance counters that report the amount of time spent executing each subtree of the execution tree, as well as a counter that reports the number of times a node was executed. Iterative application along with the Phylanx performance counters provides opportunities to take measurements and apply this gained information to future task inlining decisions.

The algorithm for task inlining in Phylanx is shown in algorithm 2. Given an iterative application, the execution time for each primitive instance is evaluated $count\_threshold$ times in order to obtain the average execution time of the primitive instance. If during the

lifetime of the application, *count_threshold* measurements are not obtained for a primitive, no decision will be made regarding the inlining of the task. If the previous primitive was executed asynchronously the next execution will be executed asynchronously. Conversely, the execution will be synchronous if the previous execution was synchronous. However, if measurements are obtained and the average execution time is below the *lower_threshold*, any future tasks created for that primitive instance will be executed synchronously and if the average execution time is above the *upper_threshold*, any future tasks created for that primitive instance will be executed asynchronously. In case where the average execution time of the primitive instance lies between the thresholds, the task will be executed with its previous mode of execution until more measurements for the execution time is gathered. The values for *count_threshold*, *lower_threshold* and *upper_threshold* are configured by the user before the start of the application.

**Algorithm 2** Task inlining policy in Phylanx

```
count_threshold                          ▷ Number of measurements to perform
lower_threshold                          ▷ Tasks below this threshold will be inlined
upper_threshold                          ▷ Tasks above this threshold will run as a separate task
exec_count ← 0                           ▷ Number of executions of the primitive instance
exec_time ← 0                            ▷ Execution time of the primitive instance
for <Every Primitive Instance> do
    if exec_count ≥ count_threshold then
        average_exec_time ← exec_time / exec_count
    end if
    if average_exec_time ≥ upper_threshold then
        inline_task ← false
    else if average_exec_time ≤ lower_thredhold then
        inline_task ← true
    else
        inline_task ← undecided
    end if
end for
```

### 4.2. Performance Impact of Task Inlining

In this section, we will look at the performance impact of task inlining on example applications in Phylanx. In order to test the effectiveness of task inlining, we use a reference

Table 4.1. Machine specifications

| **Make** | Intel | Intel | Intel | AMD |
|---|---|---|---|---|
| **Microarchitecture** | Sandybridge | Haswell | Skylake | Bulldozer |
| **CPU** | Xeon E5-2450 | Xeon E5-2660v3 | Xeon Gold 6148 | 6272 |
| **Cores** | 8 | 10 | 20 | 16 |
| **Frequency** | 2.1GHz | 2.60GHz | 2.4GHz | 2.1GHZ |

Table 4.2. Problem sizes used in LRA

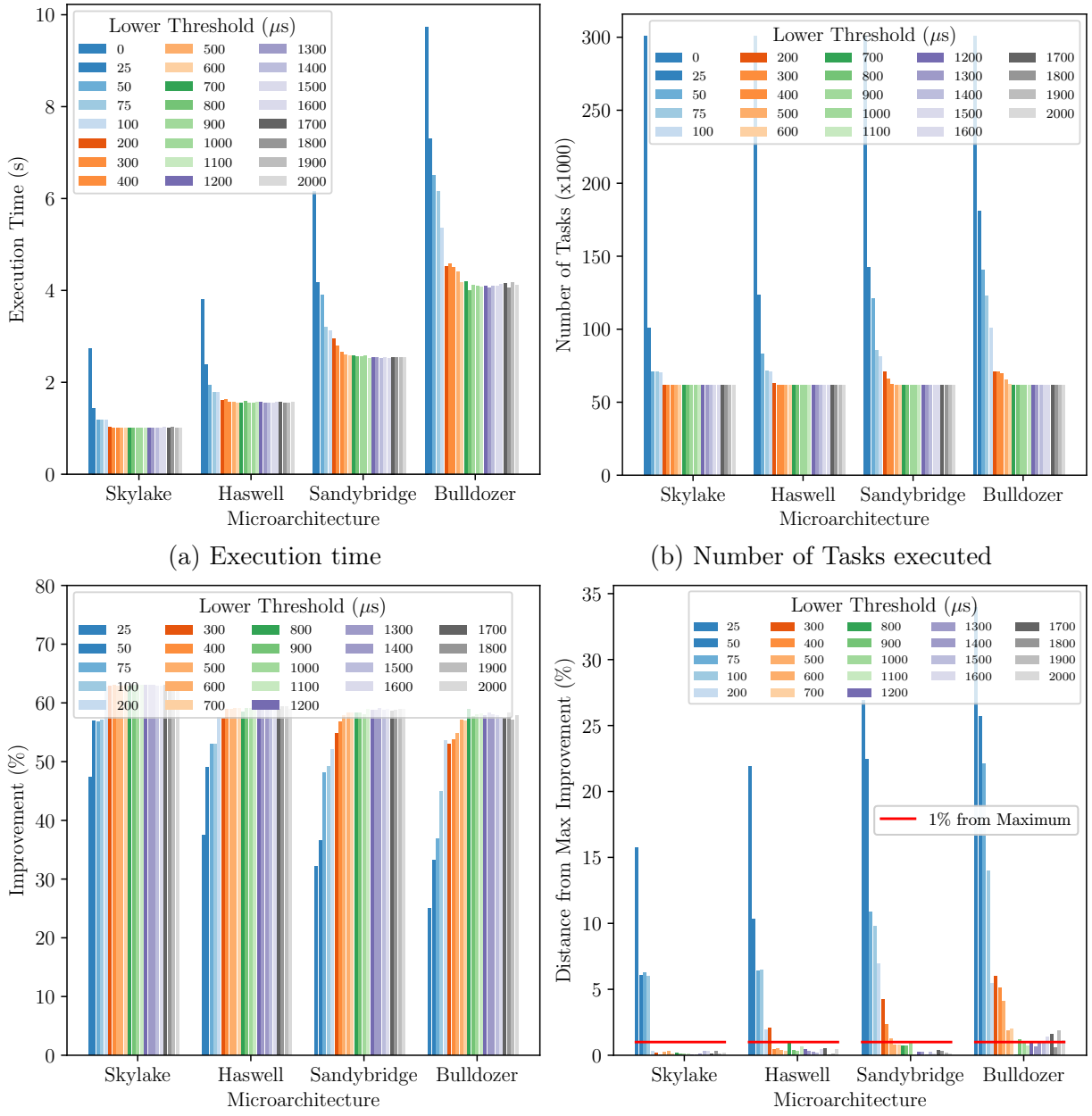
| **Problem Label** | **Iterations** | **Features** | **Observations** |
|---|---|---|---|
| LRA-P1 | 10000 | 30 | 30 |
| LRA-P2 | 10000 | 30 | 569 |

Phylanx implementation of Logistic Regression algorithm and Alternating Least Squares algorithms. Detailed description of these algorithms are presented in section 4.2.1 and 4.2.2. All experiments were performed on four different machines the specifications of which are listed in table 4.1. All the results discussed in the subsequent sections pertain to running the examples on a single thread and on eight threads. Graphs of results obtained from running the examples on four and eight threads are shown in appendix A.

### 4.2.1. Logistic Regression

The Logistic Regression algorithm [30] is a classification algorithm used in separating observations into classes. A reference implementation of the binary Logistic Regression algorithm was used for experimentation. A Python snippet of a section of the binary Logistic Regression algorithm used for experimentation in this dissertation is shown in listing 4.1. The complete Python code along with its Phylanx implementation can be found in the Phylanx Github Repository [31]. All experiments were performed on the Breast Cancer Dataset [32] with two different problem sizes labeled LRA-P1 and LRA-P2 details of which is listed in table 4.2. The execution time of the application and the total number of HPX tasks created were noted for all cases.

Figure 4.2a plots the execution time of the Logistic Regression algorithm with problem LRA-P1 (see table 4.2 for details) running on a single thread on four different processor




(a) Execution time

(b) Number of Tasks executed

(c) Improvement with respect to fully asynchronous execution

(d) Difference between current improvement and maximum improvement

Figure 4.2. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Logistic Regression example running problem LRA-P1 on one thread. All values below the red line in (d) indicate those that are within one percent of maximum improvement.

```
def lra(x, y, iterations, alpha):
weights = np.zeros(x.shape[1])
transx = np.transpose(x)
for step in range(iterations):
  g = np.dot(x, weights)
  pred = 1.0 / (1.0 + np.exp(-g))
  error = pred - y
  gradient = np.dot(transx, error)
  weights  = weights - alpha * gradient
return weights
```

Listing 4.1. Python snippet of the Logistic Regression algorithm.

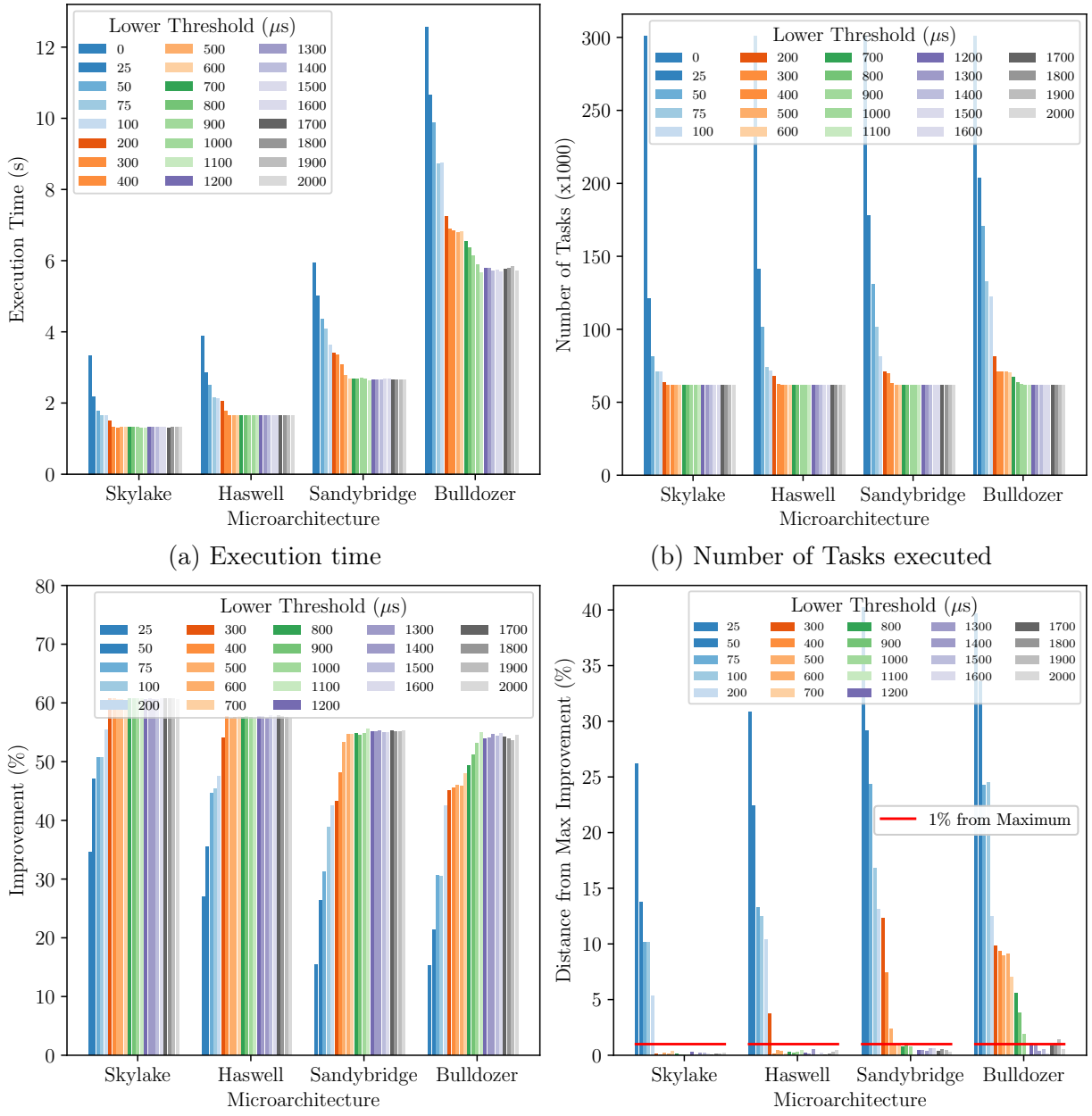
architectures namely Intel Skylake, Intel Haswell, Intel Sandybridge and AMD Bulldozer. The detailed specifications of the processors are shown in table 4.1. The application was run with various values of *lower_threshold* ranging from 0µs to 2000µs. The height of each individual bar in the figure represents execution time using a particular value for the *lower_threshold*. The value of 0µs indicates fully asynchronous execution where a new HPX task is created for every Phylanx primitive. It was observed that task inlining improved the execution time with respect to fully asynchronous execution of the test example in all four machines. Furthermore, figure 4.2c shows the percentage improvement in execution time with respect to fully asynchronous execution(*lower_threshold* set at 0µs). Increasing *lower_threshold* improves the execution time up until certain value after which it tapers off. However, it is evident that tapering off happens at different values of *lower_threshold* for different processors. For example, it can be seen that for the exact same problem, in the case of Intel Skylake machine, we do not see any improvements after increasing the *lower_threshold* beyond 200µs. However, in the case of AMD Bulldozer machine, tapering off starts at a much higher value of 600µs. However, it is seen from figure 4.2d that after tapering off, majority of *lower_threshold* values results in execution time within one percent of maximum. For the same problem, figure 4.2b shows the total number of tasks executed during the lifetime of the application for *lower_threshold* ranging from 0µs to 2000µs. When the application is run fully asynchronously (*lower_threshold*

set at 0µs), the number of tasks executed is the highest. However, as *lower_threshold* is increased, number of task executed decreases. We observe that increasing the inlining threshold increases the number of tasks being inlined which in turn decreases the overall number of tasks executed. The lowest number of tasks executed was seen with the highest value for *lower_threshold* (2000µs).

Next we look at the results of task inlining on a Logistic Regression example with the same problem(LRA-P1) with eight threads. Figure 4.3a show the execution time of the same application. Compared to the single threaded execution, running with eight threads does not results in reduction of execution time of the application. This indicates the application does not have enough parallel work in order to warrant additional threads. However, even in such a case, task inlining showed improvement in execution time of the application compared to fully asynchronous execution. Similarly, figure 4.3b shows the number of tasks created during the lifetime of the execution. As expected, higher values *lower_threshold* results in fewer tasks being created. Also, the number of tasks does not change with addition of threads. Looking at figure 4.3c , the percentage improvement in execution time of the application exhibits the same behavior as observed with the single threaded execution. It is seen from figure 4.2d that after tapering off, majority of *lower_threshold* values results in execution time within one percent of maximum as also seen with the single threaded case.

Overall, running the Logistic Regression problem LRA-P1 on both one thread and eight threads showed similar behavior. Although the application did not show improvement in execution time when additional threads were introduced, task inlining improved execution time compared to fully asynchronous execution. Furthermore, the threshold at which the performance improvement starts tapering off was not the same on all the processors but were within one percent of maximum improvement in majority of cases after tapering off.

Figure 4.4a and figure 4.4b shows the execution time and number of tasks created for single threaded execution of the Logistic Regression algorithm with problem LRA-



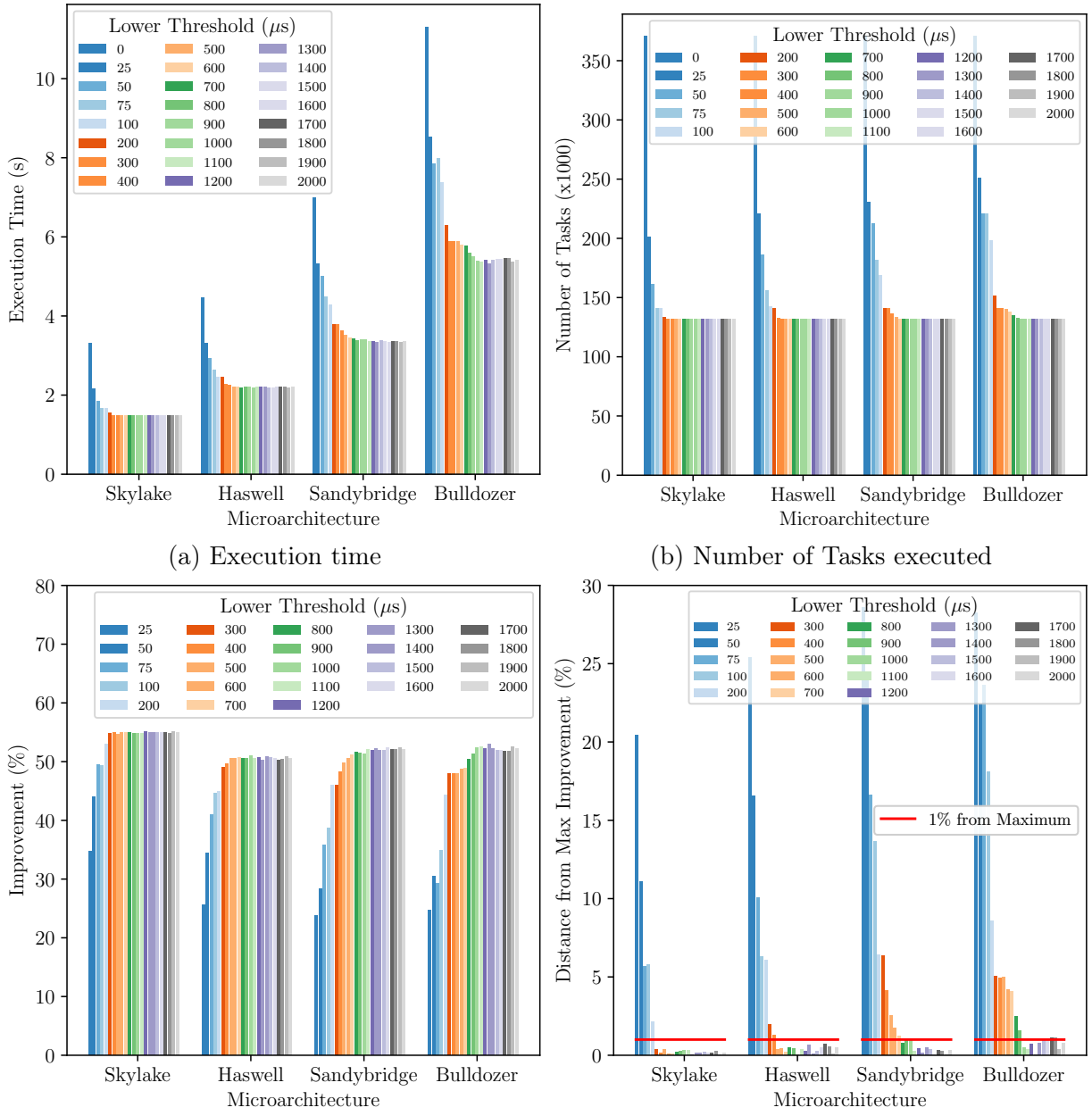

(a) Execution time

(b) Number of Tasks executed

(c) Improvement with respect to fully asynchronous execution

(d) Difference between current improvement and maximum improvement

Figure 4.3. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Logistic Regression example running problem LRA-P1 on eight threads. All values below the red line in (d) indicate those that are within one percent of maximum improvement.




(a) Execution time

(b) Number of Tasks executed

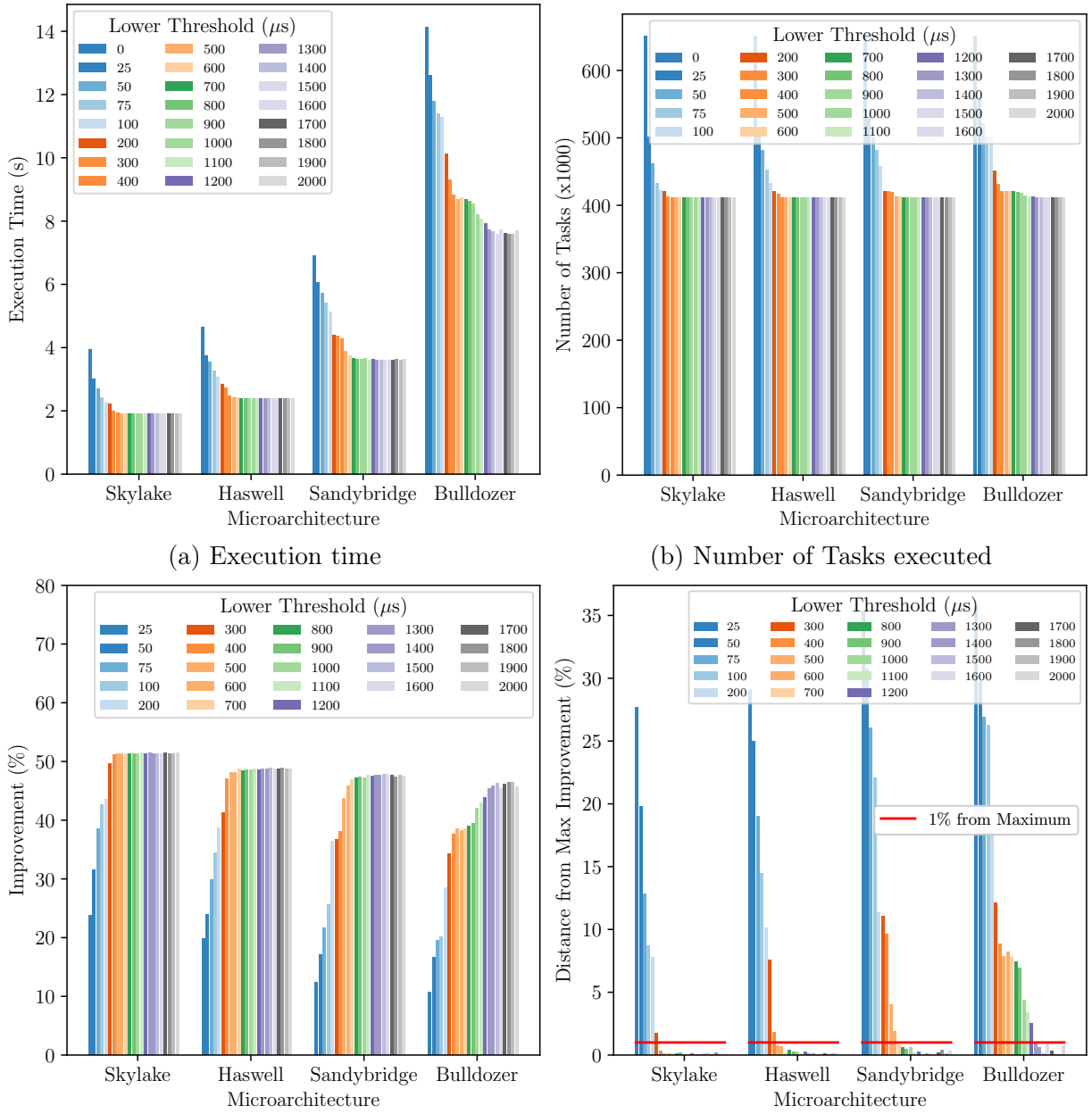
(c) Improvement with respect to fully asynchronous execution

(d) Difference between current improvement and maximum improvement

Figure 4.4. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Logistic Regression example running problem LRA-P2 on one thread. All values below the red line in (d) indicate those that are within one percent of maximum improvement.

P2. Although LRA-P2 involved larger input data, a similar behavior to LRA-P1 was noted. Referring to figure 4.4c, the tapering off of improvement in execution time after a certain value for the *lower_threshold* was also seen. We observe from figure 4.4d that after tapering off, majority of *lower_threshold* values results in execution time within one percent of maximum improvement as was the case with LRA-P1. However, for problem size LRA-P2, increasing the thread count resulted in the increase in total number of tasks executed as seen in figure 4.5b. This increase is due to the fact that larger problem size results in parallelization of matrix operations in the Phylanx toolkit. Since, Phylanx uses Blaze [33] for its matrix operations, additional tasks are created by the Blaze library to handle matrix operations. The additional tasks created by Blaze for handling matrix operations are not influenced by the inlining thresholds and are independently executed by the HPX scheduler as a new task and therefore are not inlined. This scenario allowed us to evaluate the effects of task inlining in presence of other HPX tasks. The execution time of the application as seen in figure 4.5a showed a familiar trend where a larger inlining threshold resulted in better execution times and where performance tapered off after a certain value of *lower_threshold*. Figure 4.5c shows the tapering off of improvement with respect to fully asynchronous execution and figure 4.5d shows that for most values of *lower_threshold*, improvement stays within one percent of maximum improvement.

Overall, from the experiments performed with the Logistic Regression example with the two problem sizes it was seen that task inlining improved the execution time of our test application, that improvement from task inlining tapers off after certain value for *lower_threshold* and the threshold at which improvement starts tapering off is different for different processors for the same application. Next, we observe the effects of task inlining on another application, Alternating Least Squares.




(a) Execution time

(b) Number of Tasks executed

(c) Improvement with respect to fully asynchronous execution

(d) Difference between current improvement and maximum improvement

Figure 4.5. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Logistic Regression example running problem LRA-P2 on eight threads. All values below the red line in (d) indicate those that are within one percent of maximum improvement.

```
for k in range(iterations):
  YtY = np.dot(Y.T, Y) + regularization * I_f
  XtX = np.dot(X.T, X) + regularization * I_f
  for u in range(num_users):
    conf_u = conf[u, :]
    c_u = np.diag(conf_u)
    p_u = conf_u.copy()
    p_u[p_u != 0] = 1
    A = YtY + np.dot(np.dot(Y.T, c_u), Y)
    b = np.dot(np.dot(Y.T, c_u + I_i), p_u.T)
    X[u, :] = np.dot(np.linalg.inv(A), b)

  for i in range(num_items):
    conf_i = conf[:, i]
    c_i = np.diag(conf_i)
    p_i = conf_i.copy()
    p_i[p_i != 0] = 1
    A = XtX + np.dot(np.dot(X.T, c_i), X)
    b = np.dot(np.dot(X.T, c_i + I_u), p_i.T)
    Y[i, :] = np.dot(np.linalg.inv(A), b)
return X, Y
```

Listing 4.2. Python snippet of a portion of the alternating least square algorithm.

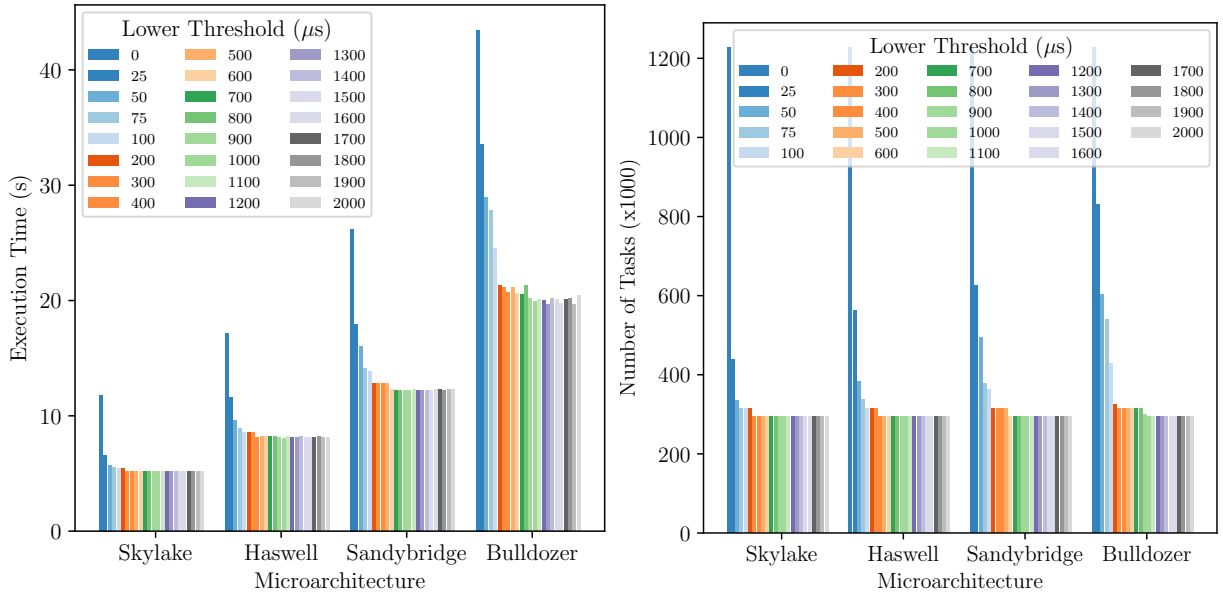
Table 4.3. Problem sizes used in ALS

| **Problem Label** | **Iterations** | **Users** | **Items** |
|---|---|---|---|
| ALS-P1 | 1000 | 10 | 10 |
| ALS-P2 | 10 | 700 | 200 |

### 4.2.2. Alternating Least Squares

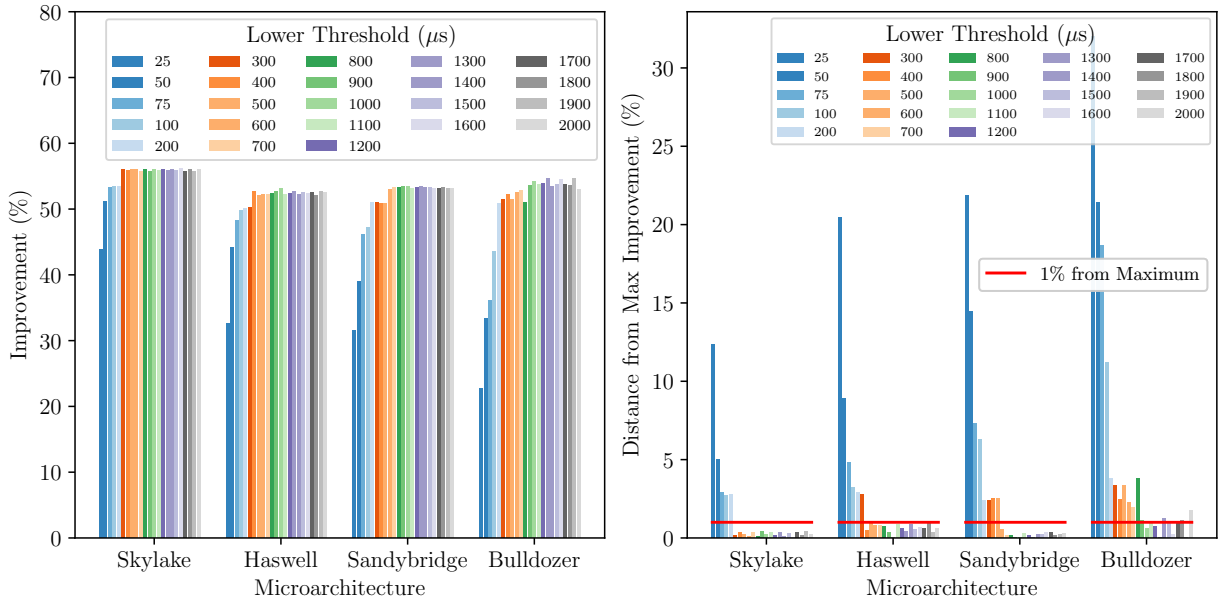
Alternating Least Squares is based on matrix factorization [34] and used in collaborative filtering. The Python snippet of a section of the Alternating Least Squares Algorithm used for experimentation in this dissertation is shown in listing 4.2. The complete code can be found in the Phylanx GitHub Repository [31]. The experiments were performed on the MovieLens Dataset [35] with problem sizes labeled ALS-P1 and ALS-P2 details for which is listed in table 4.3. The execution time of the application along with the total number of HPX tasks created were noted for all cases.

Figure 4.6a shows the execution time of the Alternating Least Squares example running




(a) Execution time

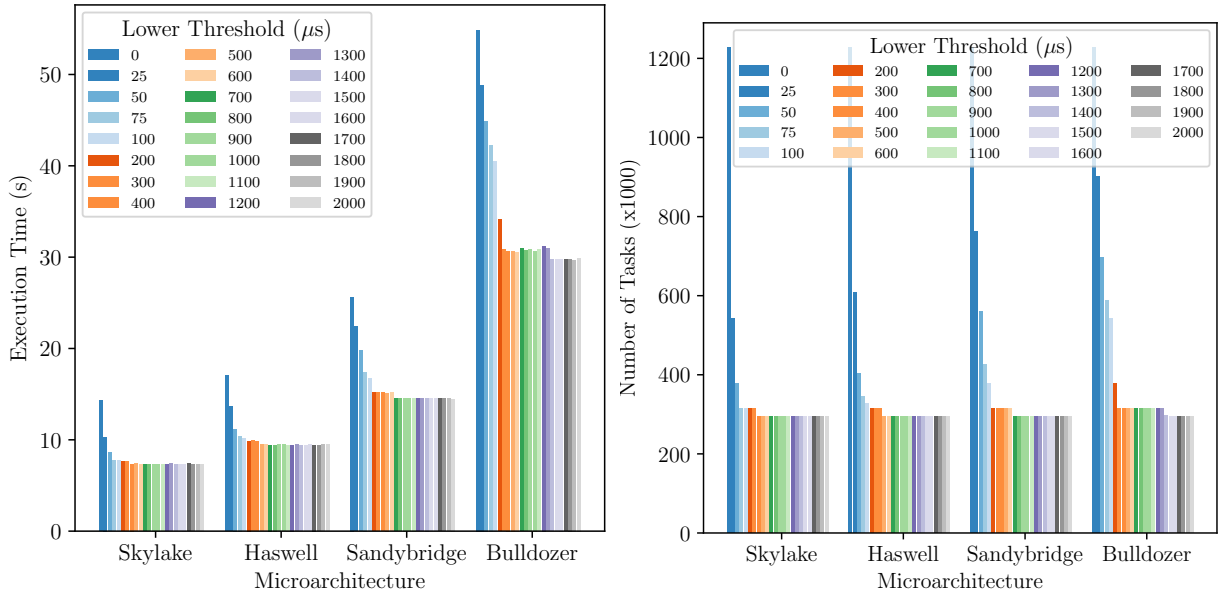
(b) Number of Tasks executed




(c) Improvement with respect to fully asynchronous execution

(d) Difference between current improvement and maximum improvement

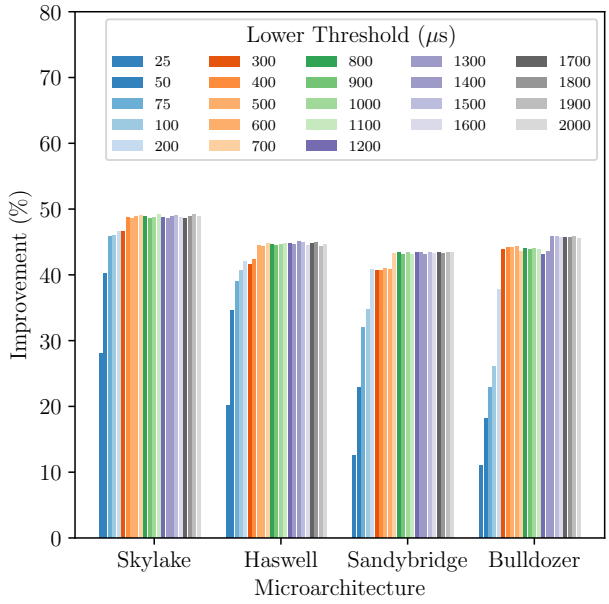
Figure 4.6. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Alternating Least Squares example running problem ALS-P1 on one thread. All values below the red line in (d) indicate those that are within one percent of maximum improvement.



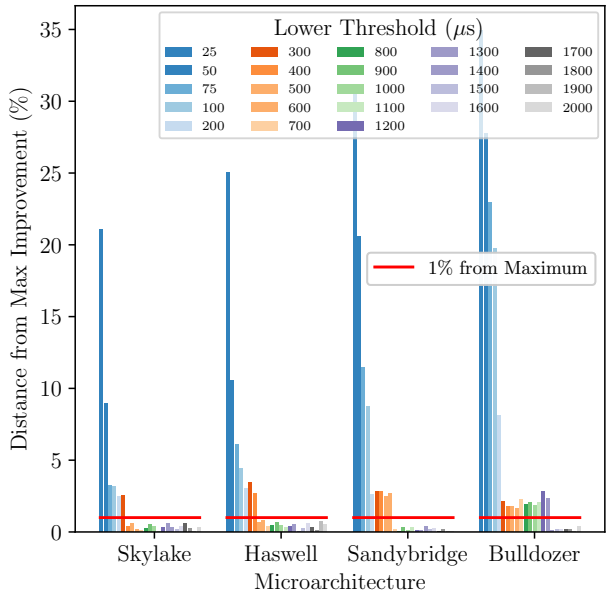

(a) Execution time




(b) Number of Tasks executed




(c) Improvement with respect to fully asynchronous execution



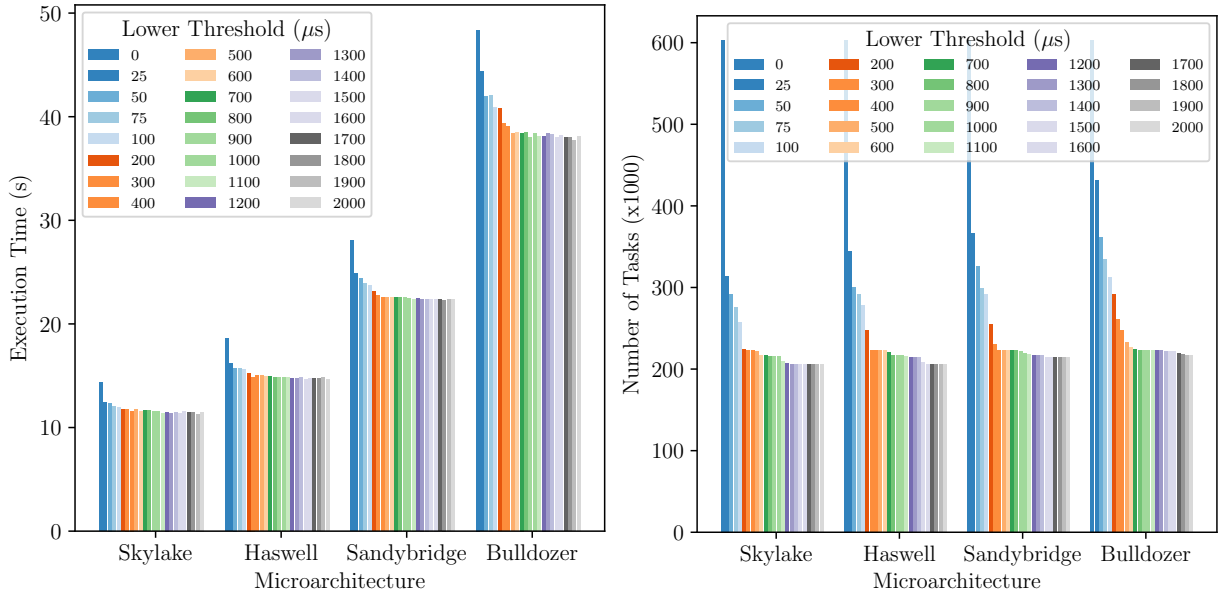

(d) Difference between current improvement and maximum improvement

Figure 4.7. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Alternating Least Squares example running problem ALS-P1 on eight threads. All values below the red line in (d) indicate those that are within one percent of maximum improvement.

problem size ALS-P1 on a single thread. In line with the Logistic Regression example, the execution time of the application is reduced when more tasks are inlined. The total number of tasks executed for the lifetime of the application for various *lower_threshold* values is shown in figure 4.6b. Fully asynchronous execution resulted in the highest number of tasks executed which gradually decreased as more tasks were inlined with higher values for *lower_threshold*. Furthermore, as seen in figure 4.6c, the improvement in execution time starts tapering off after certain values for *lower_threshold*. However, for the same problem, tapering off starts at different values of *lower_threshold* on different processors. Most values of *lower_threshold* after tapering off results in improvement within one percent of the maximum improvement (see figure 4.6d). This behavior was also seen in the Logistic Regression example. Furthermore, similar results were seen when the same example was run on eight threads (see figures 4.7a, 4.7b, 4.7c and 4.7d) .

In the next scenario, we look at the effect of task inlining on Alternating Least Squares example for problem size ALS-P2. Execution with one thread results in a similar behavior noted in previous examples where inlining improves the execution time. See figures 4.8a and 4.8b for a graphical representation of the execution time and number of tasks executed for various values of *lower_threshold*. Furthermore, as seen in Figure 4.8c and 4.8d, improvements with respect to fully asynchronous execution (*lower_threshold* set at 0µs) starts to taper off after certain value of *lower_threshold*. The value at which the improvements starts tapering off is not the same for all processors. Furthermore, in line with what was seen with previous examples, most of the values for *lower_threshold* results in improvement within one percent of maximum improvement.

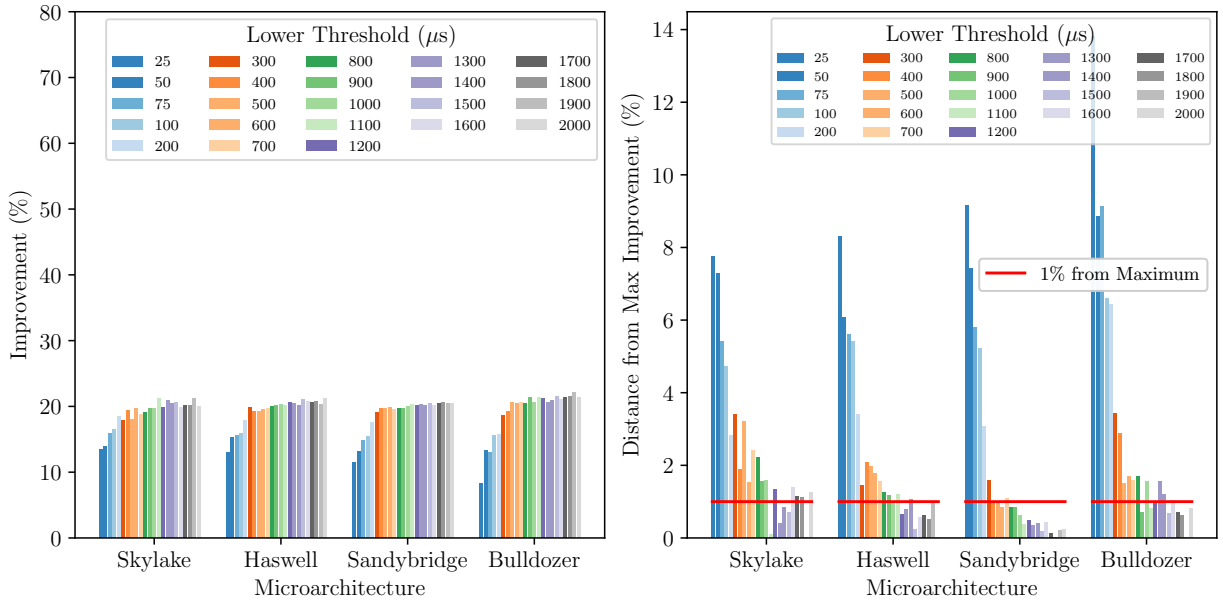
Running the same problem size with higher thread count resulted in task inlining improving the performance of the application up to a certain value of the *lower_threshold* similar to what we had seen previously. However, the number of tasks created increases when using eight threads. This increase in the total number of tasks is from the additional tasks created by Blaze for matrix operations. This can be seen in figure 4.9a and 4.9b.




(a) Execution time




(b) Number of Tasks executed

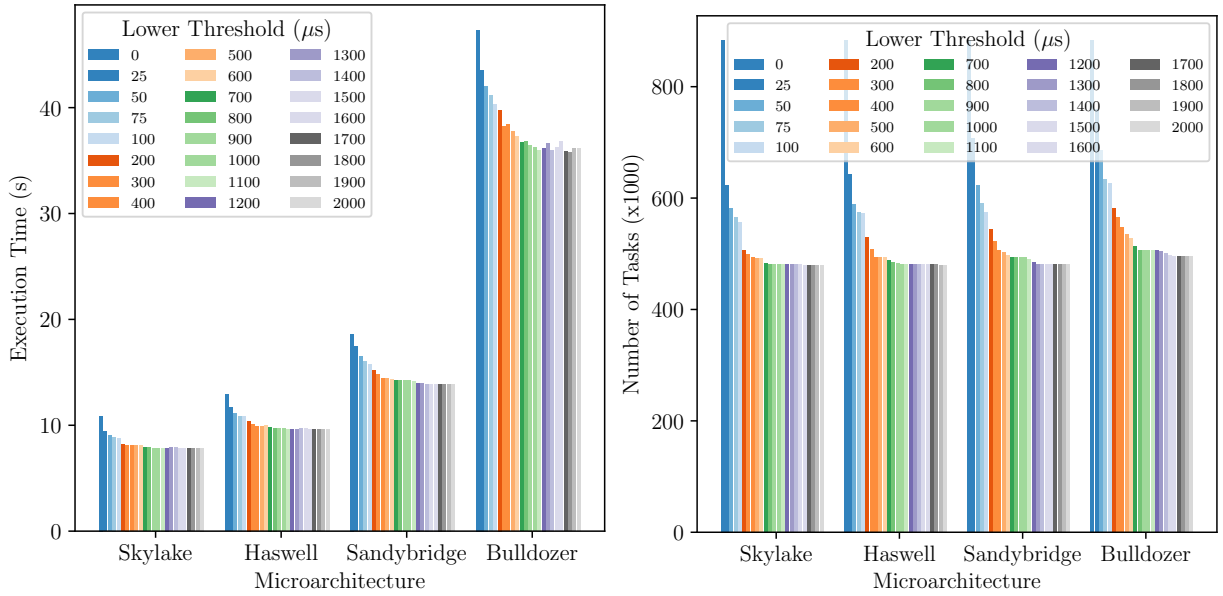



(c) Improvement with respect to fully asynchronous execution




(d) Difference between current improvement and maximum improvement

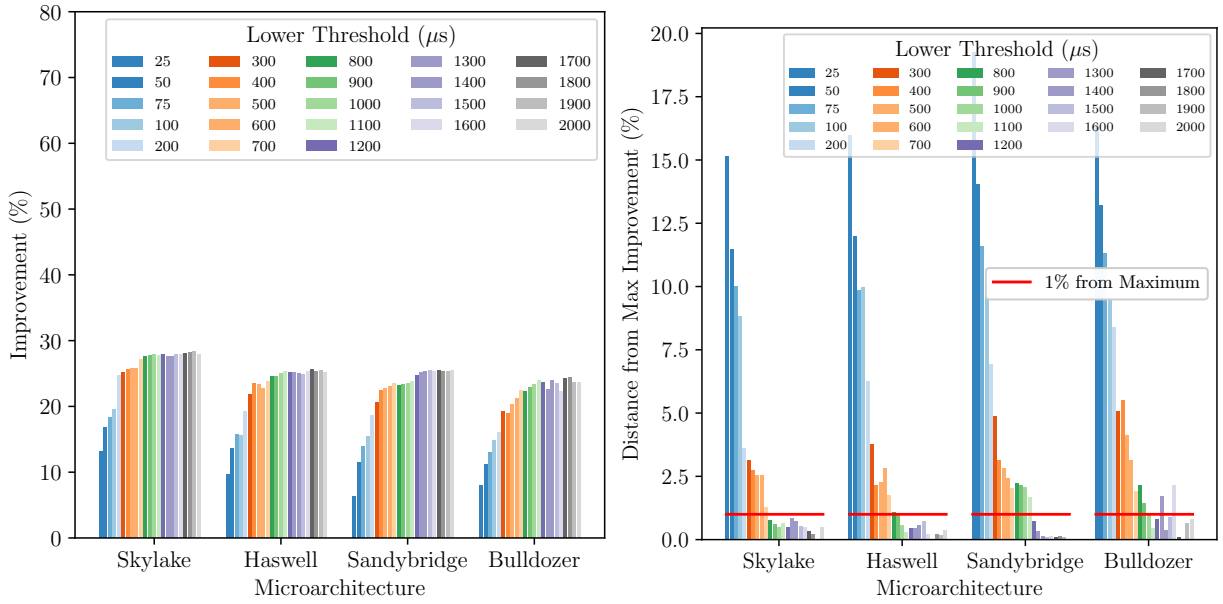
Figure 4.8. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Alternating Least Squares example running problem ALS-P2 on one thread. All values below the red line in (d) indicate those that are within one percent of maximum improvement.




(a) Execution time




(b) Number of Tasks executed




(c) Improvement with respect to fully asynchronous execution




(d) Difference between current improvement and maximum improvement

Figure 4.9. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Alternating Least Squares example running problem ALS-P2 on eight threads. All values below the red line in (d) indicate those that are within one percent of maximum improvement.

Furthermore, we see a similar behavior where improvements are tapered off after certain threshold value as seen from figure 4.9c and 4.9d. In both problem size LRA-P2 and ALS-P2, despite additional tasks being created by the Blaze library, these additional tasks are not influenced by task inlining. Furthermore, in the case of LRA-P2, adding more threads does not improve the execution time as there is not enough parallel work available in the system. In contrast, ALS-P2 improves execution time with the addition of more threads. This indicates that there is enough parallel work available in the system to keep the added cores busy.

It can be safely said that task inlining does not adversely affect the execution time of the application in any of the scenarios tested. Overall, experimentation with task inlining on the two examples for different problem sizes and architectures resulted in the following observations:

- Task inlining improves application performance compared to fully asynchronous execution.
- The improvement with respect to fully asynchronous execution starts to taper off after certain values for *lower_threshold*.
- The *lower_threshold* at which the improvement starts tapering off is different for different machines.

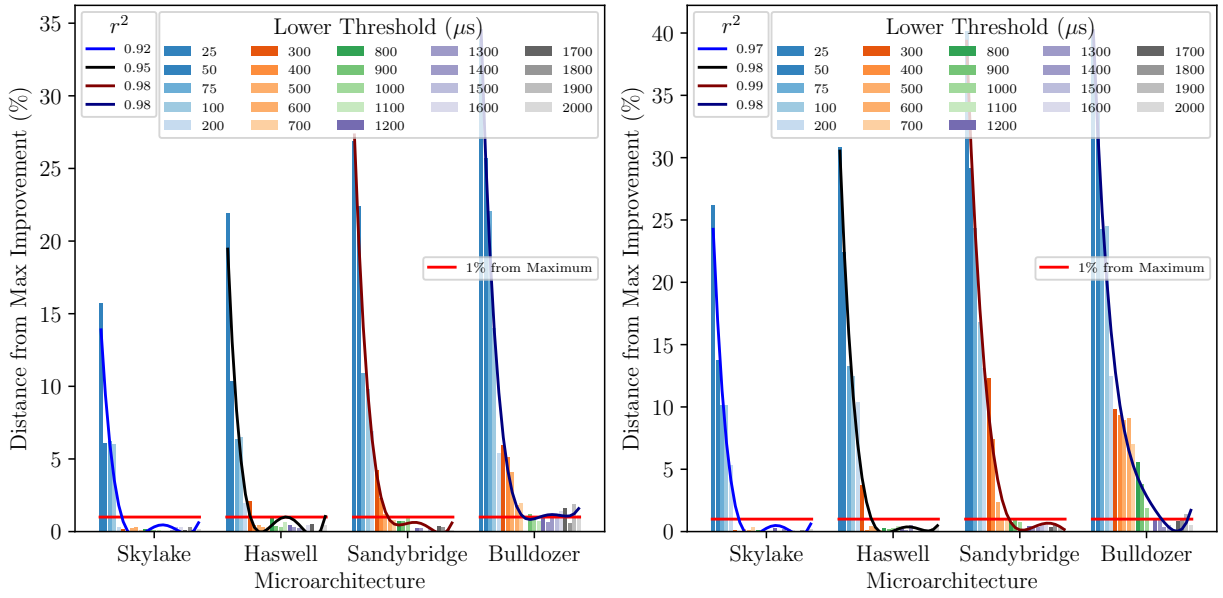
The tapering off effect is consistent with our observation with the one dimensional heat stencil experiment in section 2.2. The observation that the inlining threshold is different for different architectures allows us to take architectural effects into consideration while deciding on appropriate granularity of tasks.

### 4.3. Inlining Threshold Estimation

Asynchronous Many-Task runtime systems are founded on the idea of exploiting all available parallelism in the system by creating tasks that are small enough to utilize every clock cycle. However, since creation and management of tasks itself adds overheads there is a trade-off that needs to be balanced when deciding the task size. As an example, if the

overhead associated with creating and managing a task is equal to the amount of work contained in the task, the overall runtime of the application is effectively doubled. An important decision to make is to determine the minimum acceptable task size that amortizes the overhead cost associated with its management and creation. Experiments presented in section 2.2 where the granularity of HPX tasks were varied for a one dimensional heat stencil application showed that increasing task size beyond certain threshold did not result in performance improvement. Selecting appropriate granularity of tasks is an important factor that has large implications on application performance. Tasks that are too fine-grained results in large overheads whereas too coarse-grained tasks reduce available parallelism. If we think about task inlining from the perspective of increasing the granularity of the task via incorporating the work contained in the child task, we can take a look back into the results from section 4.2 in order make an educated guess about acceptable task size. The results from section 4.2 show that increasing the inlining threshold beyond a certain value did not further improve the application execution time. This value for the inlining threshold can be thought of as a point at which the cost of the overheads is amortized or the minimum task size necessary for the cost of overheads to be inconsequential.

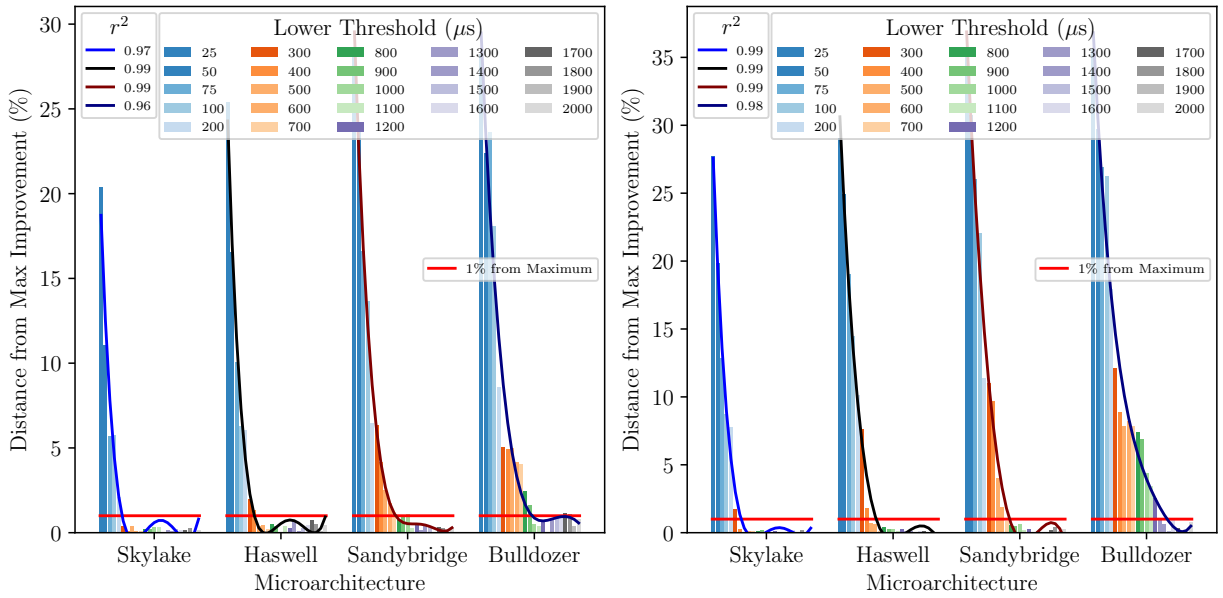
In order to determine the minimum acceptable value of the inlining threshold we now look at figure 4.10 which shows the difference between improvement (with respect to fully asynchronous execution) using current threshold value and the threshold value that attains maximum improvement for the Logistic Regression example. All values below the red line in the figure indicate those that are within one percent of maximum improvement. A polynomial regression line of degree four was fitted in the data as seen from the solid line in figure 4.10. The coefficient of determination was calculated to be above 0.90 in all the examples. The value for $lower\_threshold$ at the point of intersection of the regression line with the red line (within 1% of maximum improvement) was chosen as the minimum acceptable threshold for task inlining. Figure 4.11 shows the same for the Alternating Least Squares example. Table 4.4 summarizes the $lower\_threshold$ values for




(a) LRA-P1 on one thread




(b) LRA-P1 on eight threads

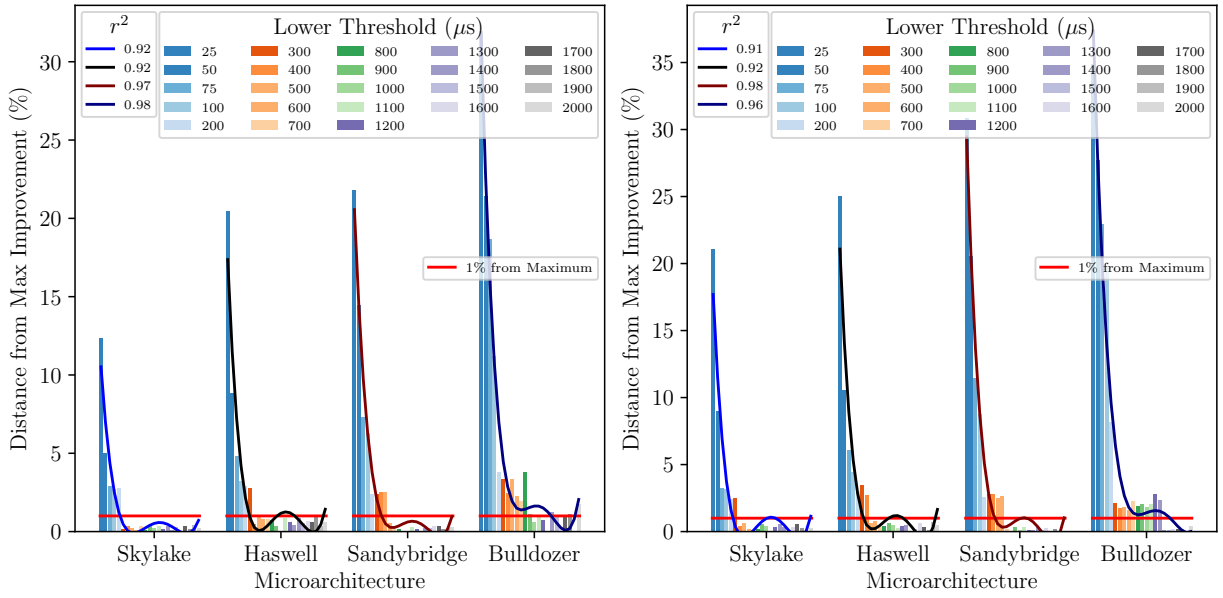



(c) LRA-P2 on one thread




(d) LRA-P2 on eight threads

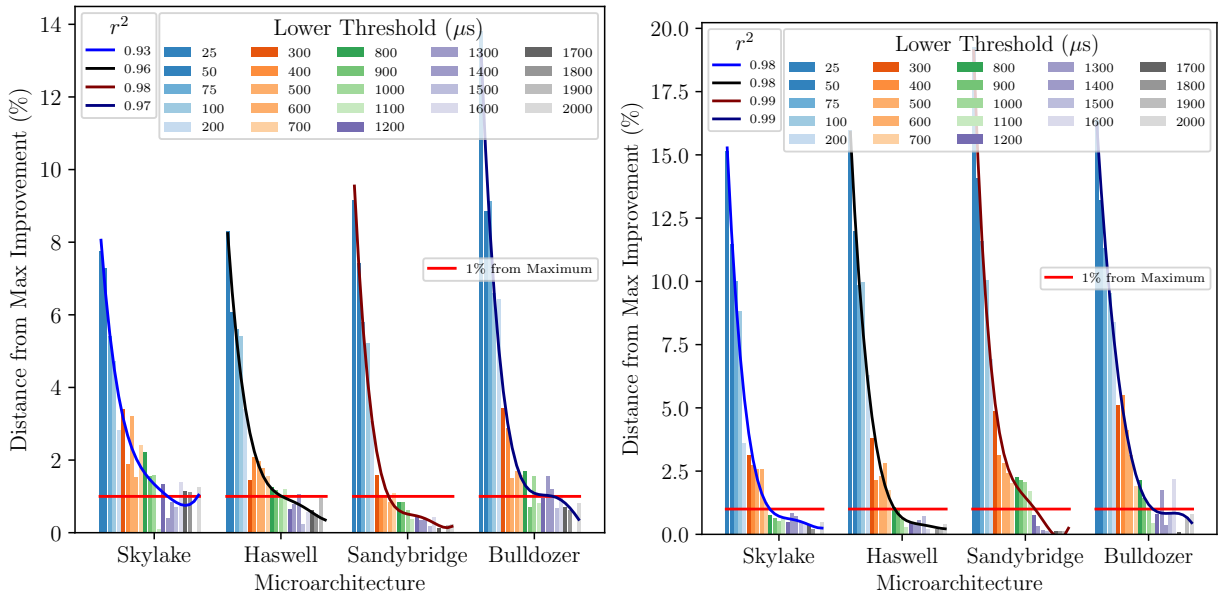
Figure 4.10. Difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Logistic Regression example along with regression line. All values below the red line indicate those that are within one percent of maximum improvement.




(a) ALS-P1 on one thread




(b) ALS-P1 on eight threads




(c) ALS-P2 on one thread




(d) ALS-P2 on eight threads

Figure 4.11. Difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Alternating Least Squares example along with regression line. All values below the red line indicate those that are within one percent of maximum improvement.

Table 4.4. Threshold (in µs) at which improvement tapers off

| | 1 Thread | | | |
|---|---|---|---|---|
| | Skylake | Haswell | Sandybridge | Bulldozer |
| ALS-P1 | 200 | 300 | 400 | 1400 |
| ALS-P2 | 1300 | 1000 | 600 | 1400 |
| LRA-P1 | 300 | 300 | 600 | 800 |
| LRA-P2 | 300 | 400 | 700 | 900 |
| | 2 Threads | | | |
| | Skylake | Haswell | Sandybridge | Bulldozer |
| ALS-P1 | 300 | 300 | 500 | 1500 |
| ALS-P2 | 1100 | 1200 | 1300 | 800 |
| LRA-P1 | 300 | 400 | 600 | 1000 |
| LRA-P2 | 300 | 500 | 700 | 1400 |
| | 4 Threads | | | |
| | Skylake | Haswell | Sandybridge | Bulldozer |
| ALS-P1 | 200 | 300 | 400 | 1500 |
| ALS-P2 | 700 | 900 | 1300 | 1100 |
| LRA-P1 | 300 | 500 | 700 | 1000 |
| LRA-P2 | 400 | 600 | 800 | 1300 |
| | 8 Threads | | | |
| | Skylake | Haswell | Sandybridge | Bulldozer |
| ALS-P1 | 200 | 300 | 400 | 1600 |
| ALS-P2 | 800 | 800 | 1200 | 1100 |
| LRA-P1 | 400 | 500 | 800 | 1400 |
| LRA-P2 | 400 | 600 | 800 | 1500 |

which improvement starts tapering off for the examples and problem sizes tested in this experimentation. Although the *lower_threshold* values presented in table 4.4 is useful in estimating acceptable inlining thresholds for the machines used for experimentation, they are not useful in estimating *lower_threshold* values on a completely different processor. Further, it would be impractical to repeat the above experiments in order to estimate the threshold each time an unseen machine is encountered.

If we think about the improvement due to task inlining from the perspective of reducing overhead costs, the larger problem of determining the appropriate inlining threshold (or the granularity of the parent task) can be distilled down to what fraction of total time spent executing a particular task is spent on overheads. Table 4.5 shows the overheads associated with creation and management of HPX futures. The results were obtained by executing

Table 4.5. Overheads per HPX-future

| Threads | Skylake | Haswell | Sandybridge | Bulldozer |
|---|---|---|---|---|
| 1 | 1.05 | 1.24 | 1.62 | 4.06 |
| 2 | 1.16 | 1.09 | 1.20 | 2.65 |
| 4 | 0.98 | 0.94 | 0.99 | 2.98 |
| 8 | 0.87 | 0.85 | 0.93 | 2.95 |

the application listed in listing 4.3. In the example, a million HPX futures were created containing no computations. The value of one million was chosen as it creates large enough number of futures to get a good estimate of overhead per future. Furthermore, the value of one million is small enough such that the calculation completes within a few seconds. The time taken to create and execute the HPX futures were measured. Even though no actual work was assigned to the tasks, housekeeping tasks such as looping over the total number of futures along with calling $push\_back()$ on the vector of futures are also included in the estimation of overheads per future. Since we are not interested in the absolute value of overheads but rather how the cost of overheads for performing the same amount of work varies with different processors, some deviation from the absolute value is acceptable. It is evident from table 4.5 that different processors results in different value for overheads of executing a HPX Future. It is seen that Skylake and Haswell machines have the lowest overheads followed by Sandybridge machine. The AMD Bulldozer machine has the highest overhead per future in the experiments performed.

### 4.3.1. Relationship Between Inlining Threshold and Task Overhead

The information seen in table 4.4 and table 4.5 show a similar trend. Skylake machines had smaller values for minimum acceptable $lower\_threshold$ and also the lowest overhead per HPX future. Similarly, AMD Bulldozer had the highest overhead per future along with highest values for the minimum acceptable $lower\_threshold$.

Let $t_{oh}$ be the overhead per HPX task and $t_{thres}$ minimum acceptable $lower\_threshold$

```cpp
// Create Action
double null_function()
{
  return 0.0;
}
HPX_PLAIN_ACTION(null_function, null_action)

// Create 1000000 futures
const std::uint64_t count = 1000000;
const hpx::id_type here = hpx::find_here();
std::vector<hpx::future<double>> futures;
futures.reserve(count);

// Start measurement
hpx::util::high_resolution_timer walltime;
for (std::uint64_t i = 0; i < count; ++i)
{
  futures.push_back(hpx::async<null_action>(here));
}
hpx::wait_all(futures);
const double duration = walltime.elapsed();

// Overhead per future in microseconds
double us = 1e6 * duration / count;
```

Listing 4.3. HPX application used for measuring per future overhead

Table 4.6. $\lambda_{min}$ values

| | 1 Thread | | | |
|---|---|---|---|---|
| | Skylake | Haswell | Sandybridge | Bulldozer |
| ALS-P1 | 190.48 | 241.94 | 246.91 | 344.83 |
| ALS-P2 | 1,238.10 | 806.45 | 370.37 | 344.83 |
| LRA-P1 | 285.71 | 241.94 | 370.37 | 197.04 |
| LRA-P2 | 285.71 | 322.58 | 432.10 | 221.67 |
| | 2 Threads | | | |
| | Skylake | Haswell | Sandybridge | Bulldozer |
| ALS-P1 | 285.71 | 275.23 | 416.67 | 566.03 |
| ALS-P2 | 948.28 | 1100.92 | 1083.33 | 301.88 |
| LRA-P1 | 258.62 | 366.97 | 500.00 | 377.35 |
| LRA-P2 | 258.62 | 458.72 | 583.33 | 528.30 |
| | 4 Threads | | | |
| | Skylake | Haswell | Sandybridge | Bulldozer |
| ALS-P1 | 204.08 | 319.15 | 404.04 | 503.36 |
| ALS-P2 | 714.29 | 957.45 | 1313.13 | 369.13 |
| LRA-P1 | 306.12 | 531.91 | 707.07 | 335.57 |
| LRA-P2 | 408.16 | 638.30 | 808.08 | 436.24 |
| | 8 Threads | | | |
| | Skylake | Haswell | Sandybridge | Bulldozer |
| ALS-P1 | 229.89 | 352.94 | 430.11 | 542.37 |
| ALS-P2 | 919.54 | 941.18 | 1209.32 | 372.88 |
| LRA-P1 | 459.77 | 588.24 | 860.22 | 474.58 |
| LRA-P2 | 459.77 | 705.88 | 860.22 | 508.47 |

Table 4.7. Average of $\lambda_{min}$ for various architectures

| | Skylake | Haswell | Sandybridge | Bulldozer |
|---|---|---|---|---|
| Mean | 465.80 | 553.11 | 662.20 | 401.53 |

beyond with improvement starts tapering off, then,

$$\lambda_{min} = \frac{t_{thres}}{t_{oh}} \tag{4.1}$$

where $\lambda_{min}$ is the minimum amount of work necessary per task in order to amortize the cost of overheads.

Table 4.6 summarizes the values of $\lambda_{min}$ for the examples tested in this study. The total number of $\lambda_{min}$ collected was 64. The mean for all 64 values was 520.66 with the

Table 4.8. Improvement within one percent of maximum with $\lambda_{min}$ set to 500

| | 1 Thread | | | |
|---|---|---|---|---|
| | Skylake | Haswell | Sandybridge | Bulldozer |
| ALS-P1 | Yes | Yes | Yes | 1.75 |
| ALS-P2 | 3.20 | 1.77 | Yes | Yes |
| LRA-P1 | Yes | Yes | Yes | 1.14 |
| LRA-P2 | Yes | Yes | Yes | Yes |
| | 2 Threads | | | |
| | Skylake | Haswell | Sandybridge | Bulldozer |
| ALS-P1 | Yes | Yes | Yes | 1.69 |
| ALS-P2 | 3.10 | 4.34 | 2.52 | Yes |
| LRA-P1 | Yes | Yes | Yes | 1.13 |
| LRA-P2 | Yes | Yes | 1.57 | Yes |
| | 4 Threads | | | |
| | Skylake | Haswell | Sandybridge | Bulldozer |
| ALS-P1 | Yes | 1.96 | 2.65 | Yes |
| ALS-P2 | 2.42 | 2.35 | 2.31 | Yes |
| LRA-P1 | Yes | Yes | 5.63 | Yes |
| LRA-P2 | Yes | 1.53 | 7.98 | Yes |
| | 8 Threads | | | |
| | Skylake | Haswell | Sandybridge | Bulldozer |
| ALS-P1 | Yes | 2.69 | 2.79 | Yes |
| ALS-P2 | 2.73 | 2.15 | 3.14 | Yes |
| LRA-P1 | Yes | Yes | 7.40 | Yes |
| LRA-P2 | Yes | 1.80 | 9.64 | Yes |

minimum being 190.47 and the maximum being 1313.13. The median was calculated to be 431.10. The mean and median values for a particular architecture is listed in table 4.7. On an average, for task inlining, the amount of work performed by the task needs be $\sim 500$ times the overhead per task. Table 4.8 shows the percentage difference from maximum improvement for all *lower_threshold* values obtained from using $\lambda_{min}$ set to 500. It is seen that for majority of cases, selecting the average $\lambda_{min}$ still resulted in performance within one percent of maximum. If a less conservative cutoff of 5% is chosen, 60 cases out of 64 results in improvement within 5% of maximum improvement. Furthermore, if $\lambda_{min}$ is set to 600, all 64 cases results in improvement within 5% of maximum improvement.

## 4.4. Summary

This chapter introduced task inlining as a means to reduce overheads associated with HPX tasks. It was seen that task inlining improved the execution time of our test application and that the improvement due to task inlining tapers off after certain value of the inlining threshold. The value at which tapering off starts is different for different processors. This chapter also developed a methodology that determines a minimal acceptable task size in the context of task inlining on HPX. Keeping the granularity of the task as small as possible is helpful as demonstrated by the experiments presented in section 2.2. The basis for deriving the minimal acceptable task size that amortizes the cost of overheads was the cost of creating and managing each individual task. The minimum acceptable task size can be derived from a single constant $\lambda_{min}$ independent of the underlying architecture of the machine. Furthermore, in chapter 5, we will use the value of $\lambda_{min}$ in other parallelization context such as such as parallel loop iteration chucking in order to estimate the size of the chunked iterations.

# Chapter 5. Loop Iteration Chunking

In chapter 4, we put forward a methodology to select the minimum granularity of HPX task in order to amortize the cost of overheads. In this chapter, we test our methodology in the context of HPX parallel loops. Chunking of parallel loop iterations is a known optimization technique where multiple iterations of a loop are combined together and executed as a single thread of execution. In order to test our methodology, we extend the existing loop chunking policy in HPX to use thresholds depending on the architecture. We present an extension of the automated chunking approach for parallel loop iterations for HPX parallel algorithms where the granularity of the combined loop iterations is derived from $\lambda_{min}$ calculated in chapter 4.

## 5.1. Loop Chunking in HPX

The static chunking policy for HPX parallel algorithms is a straightforward technique for chunking loop iterations wherein the total number of iterations is divided into equally sized chunks. One of the issues with regards to chunking loop iterations is determining the optimal chunk size. The *autochunking* policy for HPX parallel algorithms determines the chunk size of the loop iterations based on the user defined granularity of the resulting task. The policy works by executing 1% of the total number of loop iterations sequentially in order to estimate the work contained in each iteration. The disadvantage of this approach lies in the fact that only 99% of the total work is parallelized. Computing 1% of the total number of loop iterations sequentially in order to determine the the work per iteration can result in loss of parallelism. The algorithm for *autochunking* policy in HPX is given in algorithm 3.

In chapter 4, appropriate granularity for task inlining was determined on various machines. Table 5.1 lists the minimum and maximum values of task granularity in µs. In order to test whether the same values are effective in a different context, HPX *autochunking* policy is tested with the range of values listed in table 5.1.

**Algorithm 3** HPX Autochunking policy

```
min_time                                  ▷ minimum time for which a chunk should run
count                                     ▷ total number of loop iterations
test_size                                 ▷ one percent of total iterations
cores                                     ▷ number of cores used
if count > 100 x cores then
    time ← MEASURE_ONE_PERCENT()          ▷ time taken by one percent of iterations
    time_each ← time / test_size          ▷ time taken by each iteration
    if time_each >= min_time && time_each ! = 0 then
        num_iterations ← min_time / time_each
        chunk_size ← MINIMUM(count, num_iterations)
    end if
    chunk_size ← (count + cores − 1) / cores   ▷ number of iterations is less than 100 x cores
end if
```

Table 5.1. Range of threshold values in µs

| **Microarchitecture** | SandyBridge | Haswell | Skylake | Bulldozer |
|---|---|---|---|---|
| **Minimum** | 400 | 300 | 200 | 800 |
| **Maximum** | 1300 | 1200 | 1300 | 1600 |

## 5.2. Experimental Results

In order to test whether the task granularity determined in the context of task inlining would be applicable to a wider context, we utilized a toy application as well as the STREAM [36] benchmark ported to HPX. All experiments were performed on four different machines the specifications of which are listed in table 5.2.

### 5.2.1. Toy Application

In this section we present results from the toy example, a snippet of which is seen in listing 5.1. In the example, each loop iteration performed addition of an element from two vectors $a$ and $b$ and the result was stored into vector $c$. The same operation was performed with the *sequential* execution policy, the *static* chunking policy and the *autochunking* policy. All experiments were performed on four different machines the specifications of which are listed in table 5.2.

Table 5.2. Processor specifications of machines used

| **Make** | Intel | Intel | Intel | AMD |
|---|---|---|---|---|
| **Microarchitecture** | SandyBridge | Haswell | Skylake | Bulldozer |
| **CPU** | Xeon E5-2450 | Xeon E5-2660v3 | Xeon Gold 6148 | 6272 |
| **Cores** | 8 | 10 | 20 | 16 |
| **Frequency** | 2.1GHz | 2.60GHz | 2.4GHz | 2.1GHZ |

In the experiments performed with the static chunking policy, the chunk size was varied logarithmically from 100 to 100000000. The total number of loop iterations were set at 100000000 which meant that the number of chunks varied from 1000000 (when chunk size was set to 100) to 1. In order to determine the speedup obtained by executing the loop in parallel, the same application was run with the sequential policy which provided the baseline which was used to compute the speedup.

Figures 5.1 and 5.2 shows the speedup obtained using the static chunking policy for various chunk sizes on different machines. It is seen in all four cases that speedup with respect to sequential execution improves as we increase the chunk size up to a certain value after which degradation in performance is noticed. As expected, when the chunk size is set equal to the total number of iterations in the loop, the speedup is close to 1.0. The blue band in figures 5.1 and 5.2 represent the range of chunk sizes determined by the Autochunking policy fed with the range of values listed in table 5.1.

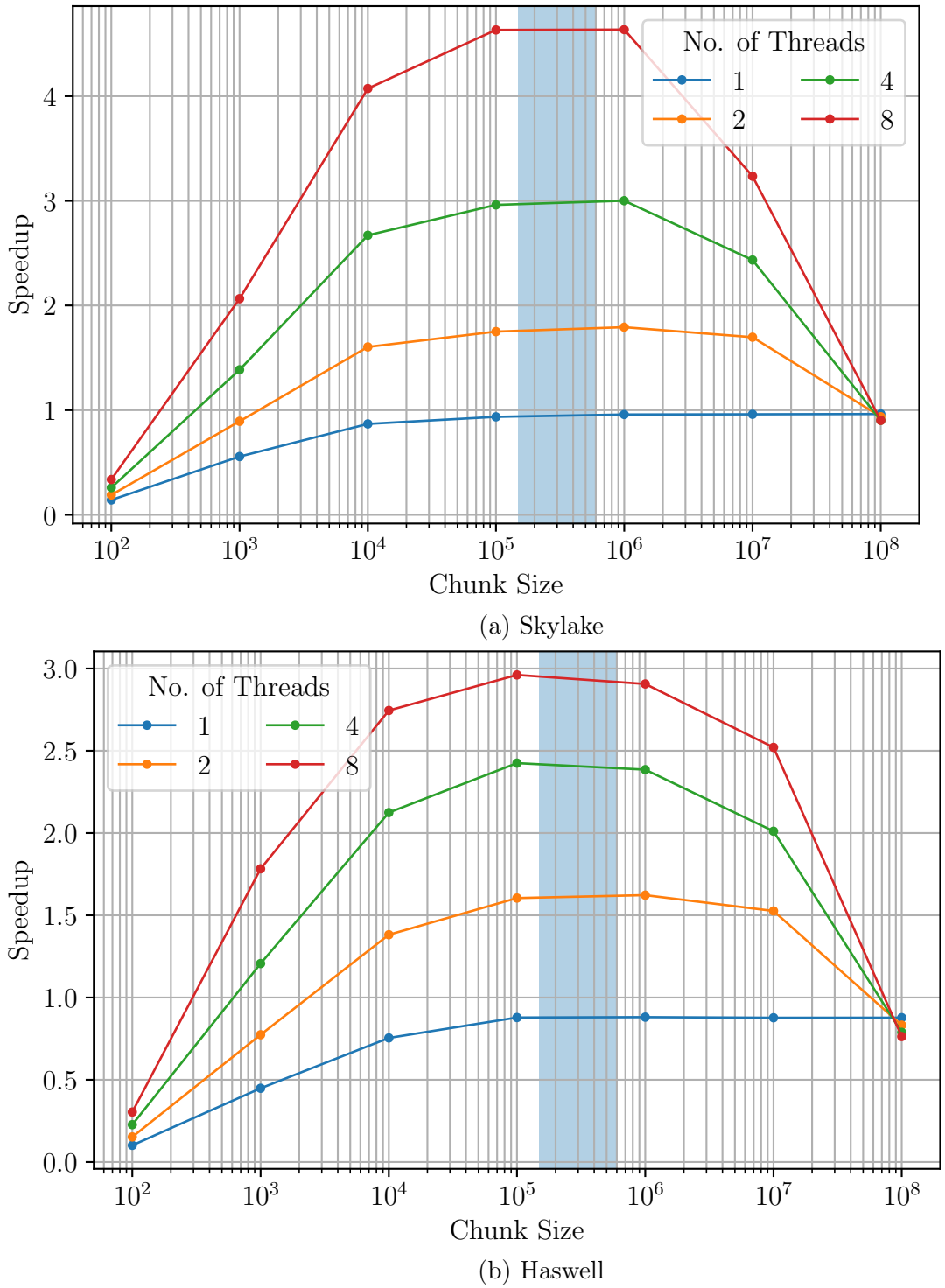
### 5.2.2. STREAM Benchmark

The STREAM [36] benchmark has been widely used for measuring memory bandwidth. The STREAM benchmark mainly consists of four operation, Copy, Scale, Add and Triad. Reference implementation of the STREAM benchmark ported into HPX, a snippet of which is seen in listing 5.2, was used as the second application for testing whether the task granularity determined in the context of task inlining would be applicable in the context of loop iteration chunking. The complete code for the HPX port of the STREAM benchmark can be found in the HPX Github repository [1]. Experiments were performed with 100000000

[1]https://github.com/STEllAR-GROUP/hpx/master/tests/performance/local/stream.cpp

```
if(policy == "acs") // Auto Chunking
  {
  std::chrono::microseconds time(acs_v);
  hpx::parallel::execution::auto_chunk_size cs(time);
  auto exec_policy = hpx::parallel::execution::par.with(cs);
  auto start = std::chrono::high_resolution_clock::now();
  hpx::parallel::for_loop(exec_policy, 0, size,
              [&](int i)
              {
              c[i] = a[i] + b[i];
              });
  auto end = std::chrono::high_resolution_clock::now();
  std::cout<<"Time:"<< std::chrono::duration<double>(end - start).count();
  }
if(policy == "scs") // Static Chunking
  {
  hpx::parallel::execution::static_chunk_size cs(scs_v);
  auto exec_policy = hpx::parallel::execution::par.with(cs);
  auto start = std::chrono::high_resolution_clock::now();
  hpx::parallel::for_loop(exec_policy, 0, size,
              [&](int i)
              {
              c[i] = a[i] + b[i];
              });
  auto end = std::chrono::high_resolution_clock::now();
  std::cout<<"Time:"<< std::chrono::duration<double>(end - start).count();
  }
if(policy == "seq") // Sequential Execution
  {
  auto exec_policy = hpx::parallel::execution::seq;
  auto start = std::chrono::high_resolution_clock::now();
  hpx::parallel::for_loop(exec_policy, 0, size,
              [&](int i)
              {
              c[i] = a[i] + b[i];
              });
  auto end = std::chrono::high_resolution_clock::now();
  std::cout<<"Time:"<< std::chrono::duration<double>(end - start).count();
  }
```

Listing 5.1. Snippet of the toy application employing HPX parallel_for

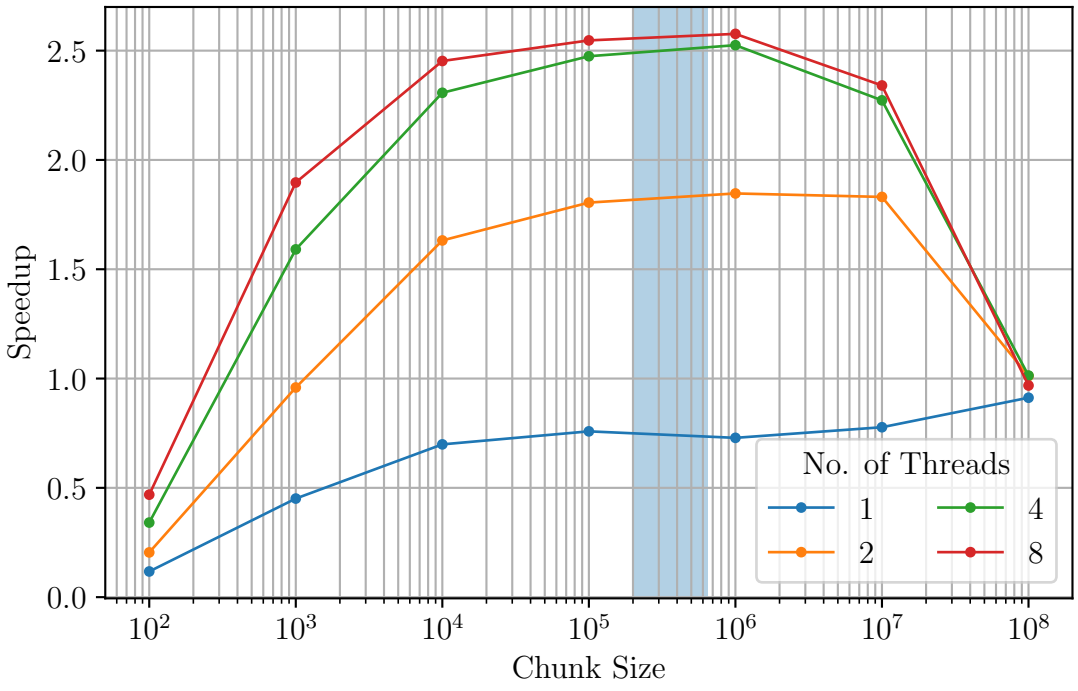



(a) Skylake

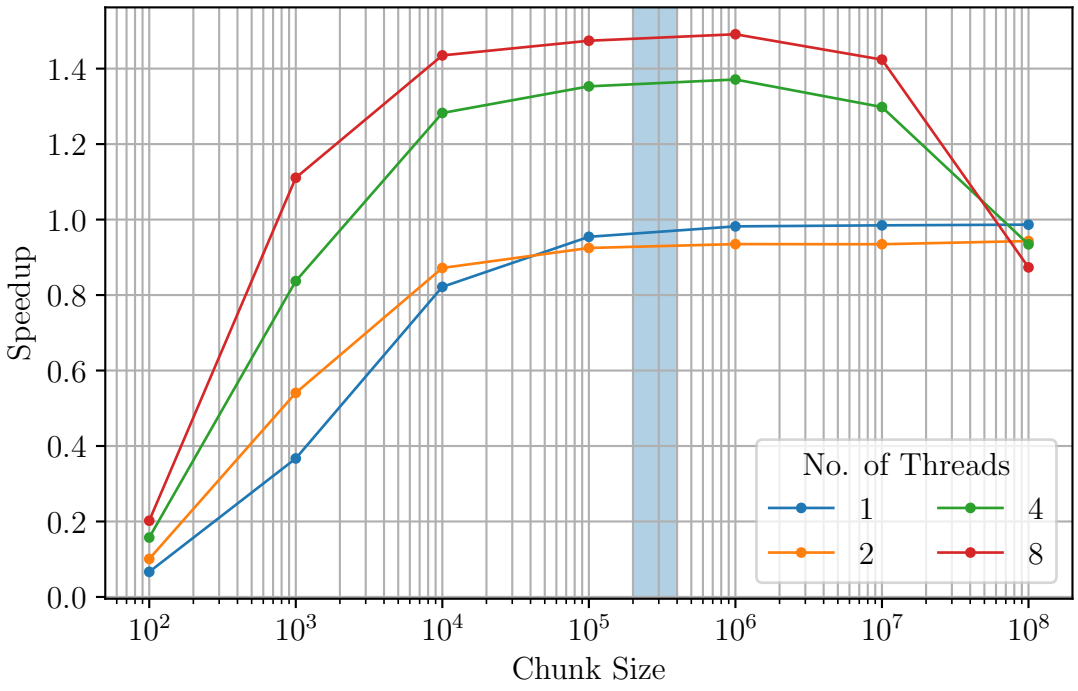



(b) Haswell

Figure 5.1. Sweep of chunk sizes for various threads on (a) Skylake and (b) Haswell machines using the static chunking policy in HPX. The blue band in the figure represent the range of chunk sizes obtained using the Autochunking policy.




(a) Sandybridge




(b) Bulldozer

Figure 5.2. Sweep of chunk sizes for various threads on (a) Sandybridge and (b) Bulldozer machines using the static chunking policy in HPX. The blue band in the figure represent the range of chunk sizes obtained using the Autochunking policy.

```
template <typename T>
struct multiply_step
{
  multiply_step(T factor) : factor_(factor) {}
  HPX_HOST_DEVICE HPX_FORCEINLINE T operator()(T val) const
  {
    return val * factor_;
  }
  T factor_;
};

template <typename T>
struct add_step
{
  HPX_HOST_DEVICE HPX_FORCEINLINE T operator()(T val1, T val2) const
  {
    return val1 + val2;
  }
};

template <typename T>
struct triad_step
{
  triad_step(T factor) : factor_(factor) {}
  HPX_HOST_DEVICE HPX_FORCEINLINE T operator()(T val1, T val2) const
  {
    return val1 + val2 * factor_;
  }
  T factor_;
};

for(std::size_t iteration = 0; iteration != iterations; ++iteration)
  {
  // Triad
  timing[3][iteration] = mysecond();
  hpx::parallel::transform(policy,
    b.begin(), b.end(), c.begin(), c.end(), a.begin(),
    triad_step<STREAM_TYPE>(scalar)
    );
  timing[3][iteration] = mysecond() - timing[3][iteration];
  }
```

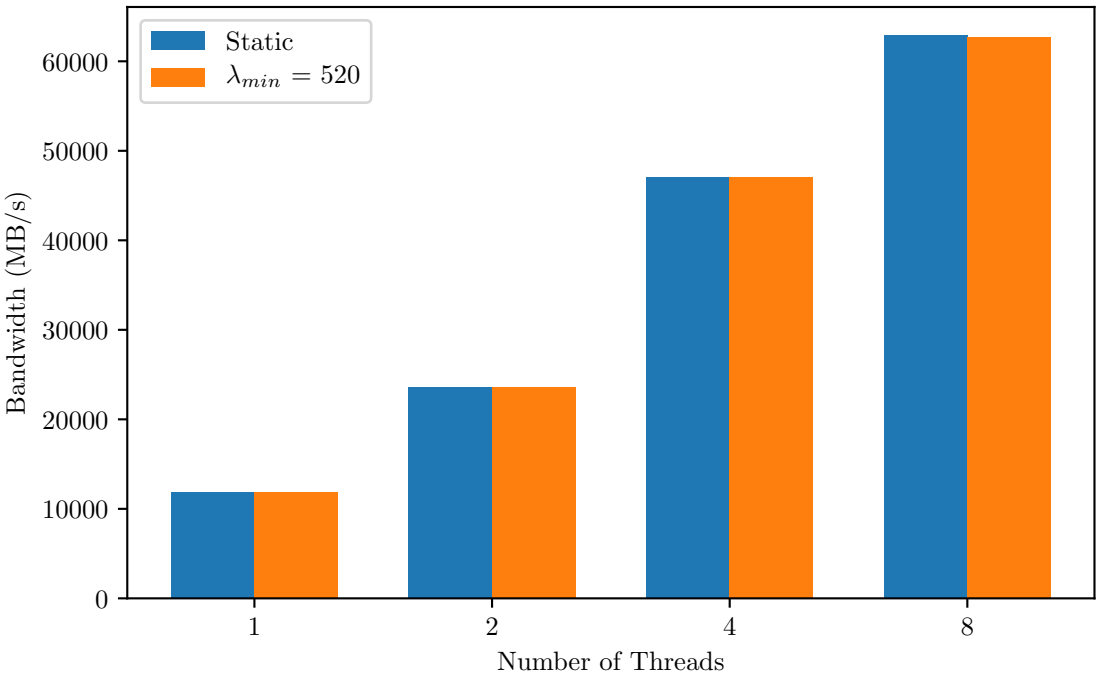
Listing 5.2. Snippet of STREAM benchmark ported to HPX showing the TRIAD step

elements. We ran the STREAM benchmark with *static* chunking policy resulting in the loop iterations being divided equally among the processing units. Furthermore, we ran the same benchmark with Autochunking policy where the granularity of combined loops were determined by setting $\lambda_{min}$ to 520 which was the average value of $\lambda_{min}$ for all 64 cases of task inlining experiments.

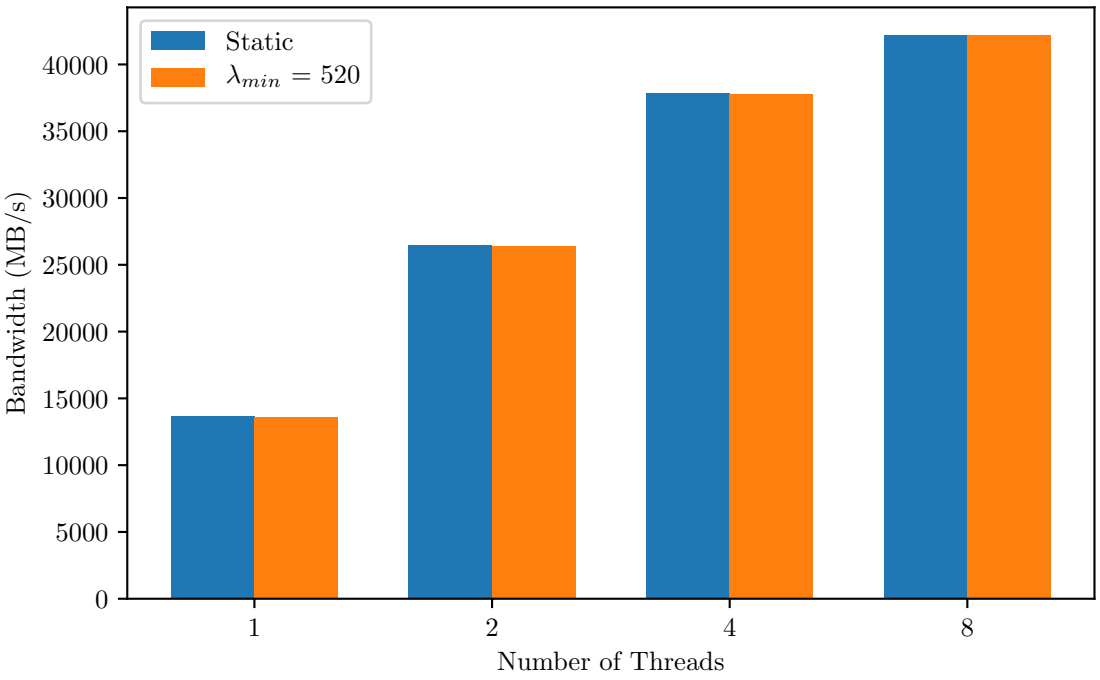
Figure 5.3 and figure 5.4 shows the memory bandwidth obtained for the STREAM TRIAD on Skylake, Haswell, Sandybridge and Bulldozer machines. It is seen that deriving task granularity from $\lambda_{min}$ resulted in performance comparable to the ones obtained by equally dividing the total work among the processing cores.

## 5.3. Summary

In this chapter, we utilized the minimum task granularity determined in the context of task inlining in a different context viz. chunking of loop iterations. Using $\lambda_{min}$ calculated in chapter 4, it is possible to estimate the task granularity of a particular machine in a setting other than task inlining. Furthermore, for the applications and examples tested, the performance of the application was comparable even when the granularity of the task was reduced compared to statically dividing the total work equally among the processing units.

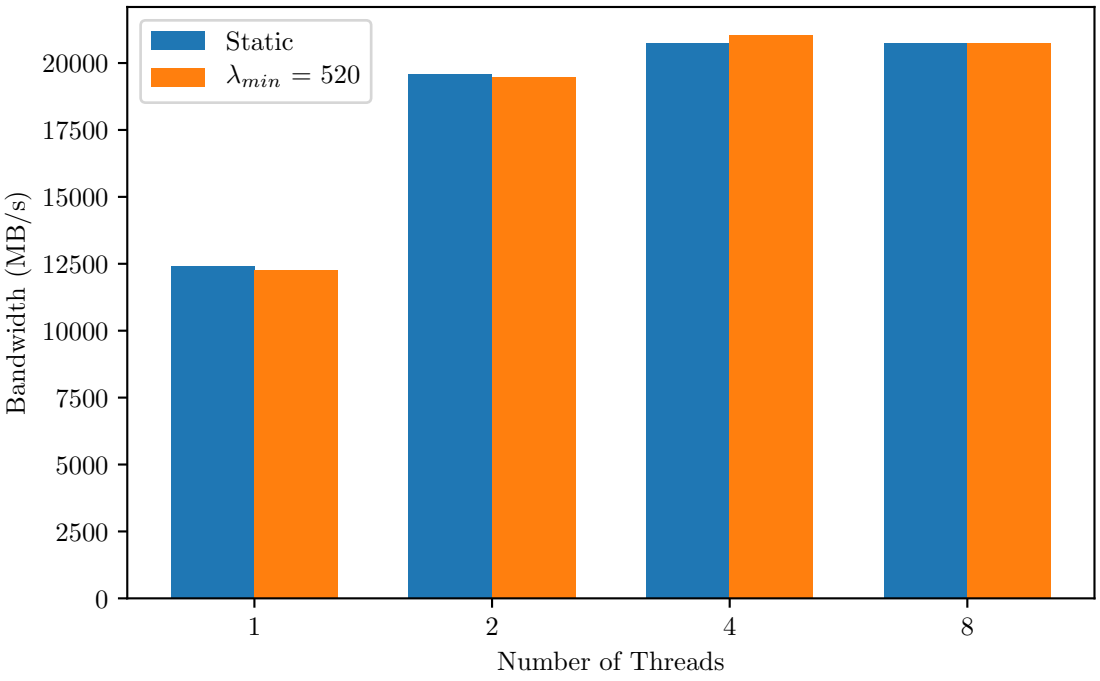



(a) Skylake

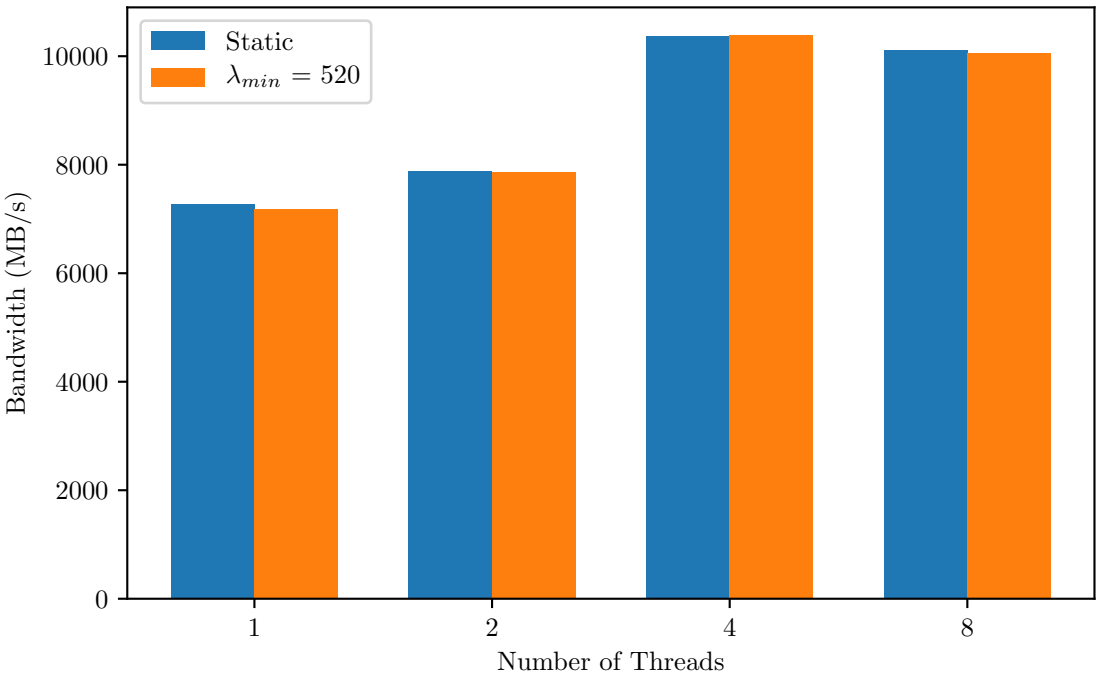



(b) Haswell

Figure 5.3. Memory bandwidth obtained from running the STREAM TRIAD benchmark on (a) Skylake and (b) Haswell with static chunking policy with the work being divided equally among the cores and using chunksize obtained by setting $\lambda_{min}$ at 520




(a) Sandybridge




(b) Bulldozer

Figure 5.4. Memory bandwidth obtained from running the STREAM TRIAD benchmark on (a) Sandybridge and (b) Bulldozer with static chunking policy with the work being divided equally among the cores and using chunksize obtained by setting $\lambda_{min}$ at 520

# Chapter 6. Related Work

This dissertation studies task overheads in HPX, an Asynchronous Many-Task runtime system. In this chapter, recent research related to this work is presented. Related works can be broadly summarized as those that pertain to selecting appropriate task granularity and message size with regards to Asynchronous Many-Task runtime systems.

OpenMP [4,37] has been a popular parallel programming tool along with MPI [3]. Recently, However, task based programming models and runtime systems have started to carve out its own space in the parallel computing realm. HPX and Charm++ [20] are examples of asynchronous task based runtime systems. Chapel [38] and X10 [39] are programming language based solutions that provide the notion of tasks. Cilk [40], Intel TBB [41] and Legion [42] are some more examples of solution that perform parallel computation using tasks.

Mohr et al. in [27] used task inlining to control the granularity of tasks on Mul-T [43], a parallel implementation of Scheme. Two strategies were considered, the first one being a $load - based\ inlinining$ strategy where a task was inlined if the system load was beyond a certain threshold. The second strategy considered was *lazy task creation* where task creation was avoided until processing resources were free. The work presented in this dissertation incorporates task inlining in the context of Asynchronous Many-Task runtime systems and makes inlining decisions regarding a particular task based on previous execution time of the same task.

With regards to OpenMP tasks, Duran et al. in [44] evaluated two scheduling approaches: $breadth - first$ and $work - first$ approach on the NANOS research OpenMP runtime. Furthermore, a cut-off technique was proposed in order to improve performance. The cutoff was based on either the max number of tasks in the system or max task recursion level. A continuation of the same research resulted in ATC(Adaptive Task Cutoff), proposed in [45], where the cutoff decision was based on the profiling data obtained from the application at runtime. The profiler was implemented on the NANOS runtime. OpenMP

tasks were executed by nano-tasks which are user level NANOS threads running on top of POSIX threads. After gathering profiling information, computational load of the task is estimated. The task is only created if the estimated grain size is larger than 1 millisecond. Adaptive Task granularity(ATG) was proposed in [46] for irregular task parallel programs. ATG switches between help first and serialization policy depending upon the number of tasks created in the system. However, the effects of processor architectures were not considered.

With regards to compiler based approaches, Thoman et al. in [47] proposed a combined runtime as well as compiled time approach for automatically controlling the granularity. Here, multiple versions of each parallel tasks with varying granularity was generated at compile time. The runtime system then made a selection from these tasks of varying granularity by looking at task demand. Iwasaki and Taura in [48] proposed a static cutoff technique by identifying a condition when recursion stops at which point task creation is eliminated. Furthermore, two optimization methods are proposed namely $code - bloat - free\ inlining$ where expansion of subproblem calls are inlined and $loopification$ where tasks are transformed into loops that can be vectorized. Again, Iwasaki and Taura in [49] proposed an auto-tuning framework for divide and conquer task parallel programs. The framework searches for optimal combination of transformation methods outlined in [48] and was implemented as an optimization pass in LLVM. Our methodology for estimating the granularity of inlined task does not require additional compilation steps and no change in application source code is required.

Akhmetova et al. studied the interplay between task granularity and scheduling overhead [50]. Experiments were performed on the Prometheus emulation tool using applications written in Cilk. Optimal task granularity was found to be between $1.2\text{x}10^4$ and $10\text{x}10^4$ cycles. Sun [51] developed the ParSSSE (Parallel State Space Search Engine) Framework for Charm++ and looked at adaptive grain size control in the context of parallel state search methods. In the startup phase, on each node fine grained tasks are created in order

to saturate the processors available. Once saturation is reached, medium grained tasks are created in order to minimize the overheads. Furthermore, adaptively determining task granularity was also incorporated into the ParSSSE framework. Grubel [23], used performance counters in HPX for dynamically tuning grain size of 1d-stencil application. Recent work by Sutterlein [52] extended the Roofline [53] model for task based runtime systems and applied the model for determining granularity of task on P-OCR [54] runtime system.

The work presented in this dissertation takes into consideration the effects of processor architecture on the granularity of inlined tasks. Furthermore, our proposed method is application agnostic and no change in application code is required. Our method relies on the actual execution time of the the tasks in order to make decisions regarding task inlining for future execution of the tasks at runtime. Furthermore, we estimate the minimum granularity of a task using a single largely machine independent constant which was shown to be useful in not just task inlining but also in other parallelism context such as chunking of loop iterations.

Combining many small messages and sending them as a larger message is an optimization technique that has been in use for quite some time now [55]. Other task based runtime systems that have the ability to coalesce messages include Active Pebbles [18], AM++ [19] and Charm++ [25]. Our implementation of message coalescing uses number of individual messages to coalesce in one larger message as a means of deciding when a message is sent whereas Active Pebbles, AM++ and Charm++ use buffer size for the same. Active Pebbles and AM++ sends the message when the buffer is full or and also supports a flush method for immediate send. Charm++ has a periodic check mechanism which performs an immediate send if no messages where send between subsequent checks. Our implementation of message coalescing allows the coalesced message to be sent after a timeout. When the first message enters the coalescing queue, a timer is set which flushes the coalescing queue on expiration of the timer. Hence, each instance of coalesced messages is sent out either when the coalescing queue is full or when the timeout is triggered. With regards to adap-

tive approach for message coalescing, TRAM [25] has successfully demonstrated a basic approach where different sets of parameters for coalescing are tried during each iterative step on an all-to-all benchmark using PICS: A Performance-Analysis-Based Introspective Control System [26]. Metrics identified in this research have shown a strong correlation with execution time of the test applications and can aid in evaluating network efficiency by giving us an intrinsic view of the underlying network overhead which would be difficult to measure using conventional methods and can be useful as a basis for the adaptive tuning of a broad set of message coalescing parameters.

# Chapter 7. Conclusion

This work focused on methodologies for managing overheads in the context of Asynchronous Many-Task runtime systems using HPX as an exemplar implementation. Overheads were categorized into broadly two categories: overheads associated with local task execution and overheads associated with tasks executed remotely. Three existing techniques were applied to the HPX runtime system viz. active message coalescing, task inlining and parallel loop iteration chunking with the goal of lessening the effects of overheads.

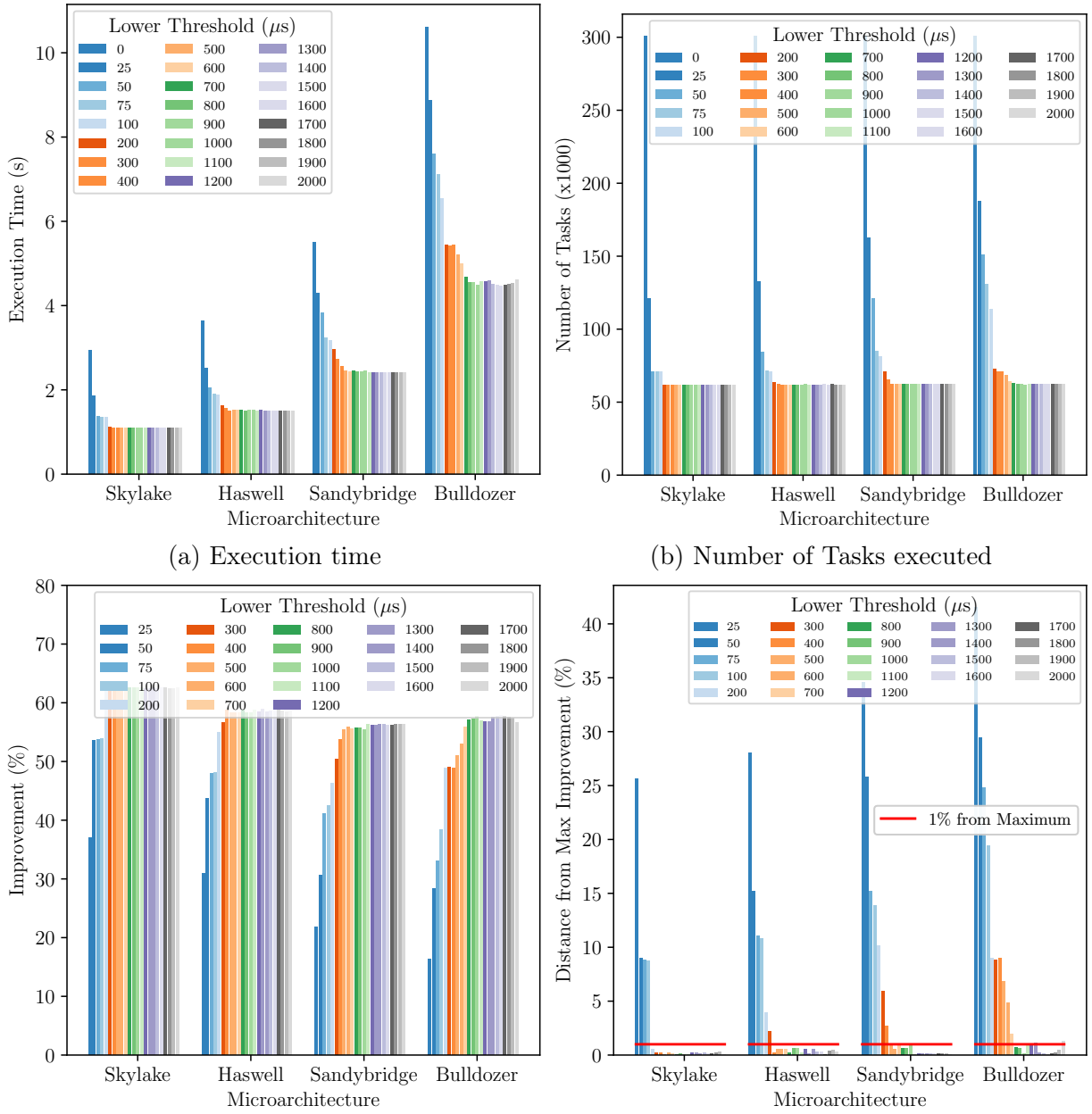
With regards to overheads associated with tasks executed locally, a dynamic policy that decides whether to inline a particular task at runtime was developed. It was seen that the dynamic policy was able to attain an improvement of up to 63% over fully asynchronous execution for the examples tested in this work. Also in the context of task inlining, a methodology for estimating the granularity of task in relation to the overheads associated with its creation and management was also developed. It was seen that task granularity is different for different processor architectures. A largely architecture independent constant, $\lambda_{min}$, based on the overheads associated with creation and management of tasks was derived. In the context of HPX parallel loop chunking, the same constant was used to to estimate the chunk size of HPX parallel loops.

With regards to overheads associated with executing a task remotely, effects of active message coalescing in the context of HPX was studied. Furthermore, methodologies and metrics for analyzing the overheads associated with transmission and reception of active messages were developed. The metrics identified in this research aid in evaluating network efficiency by giving us an intrinsic view of the underlying network overhead which would be difficult to measure using conventional methods.

In the future, with regards to the metrics pertaining to active message coalescing, the strong positive correlation between execution time and network overhead opens new possibilities for advanced adaptive solutions for message coalescing. The runtime system could tune its coalescing parameters dynamically by evaluating the overhead counters provided

by the performance counter framework. In the future, metrics and techniques defined in this research could be used as a basis for the adaptive tuning of a broad set of messaging parameters. With regards to task inlining, the effect of task inlining on GPUs would be also be an avenue for further research. With regards to the *autochunking* policy in HPX, one of the disadvantage of the policy is the fact that in order to estimate the granularity of chunked loop iterations, one percent of the loop is executed in serial. This can be improved in the future by reusing the estimated granularity for a particular loop if the loop is executed more than once. Hence, for loops that are executed multiple times, the one percent serial execution penalty is only felt once.

# Appendix A. Supplementary Results



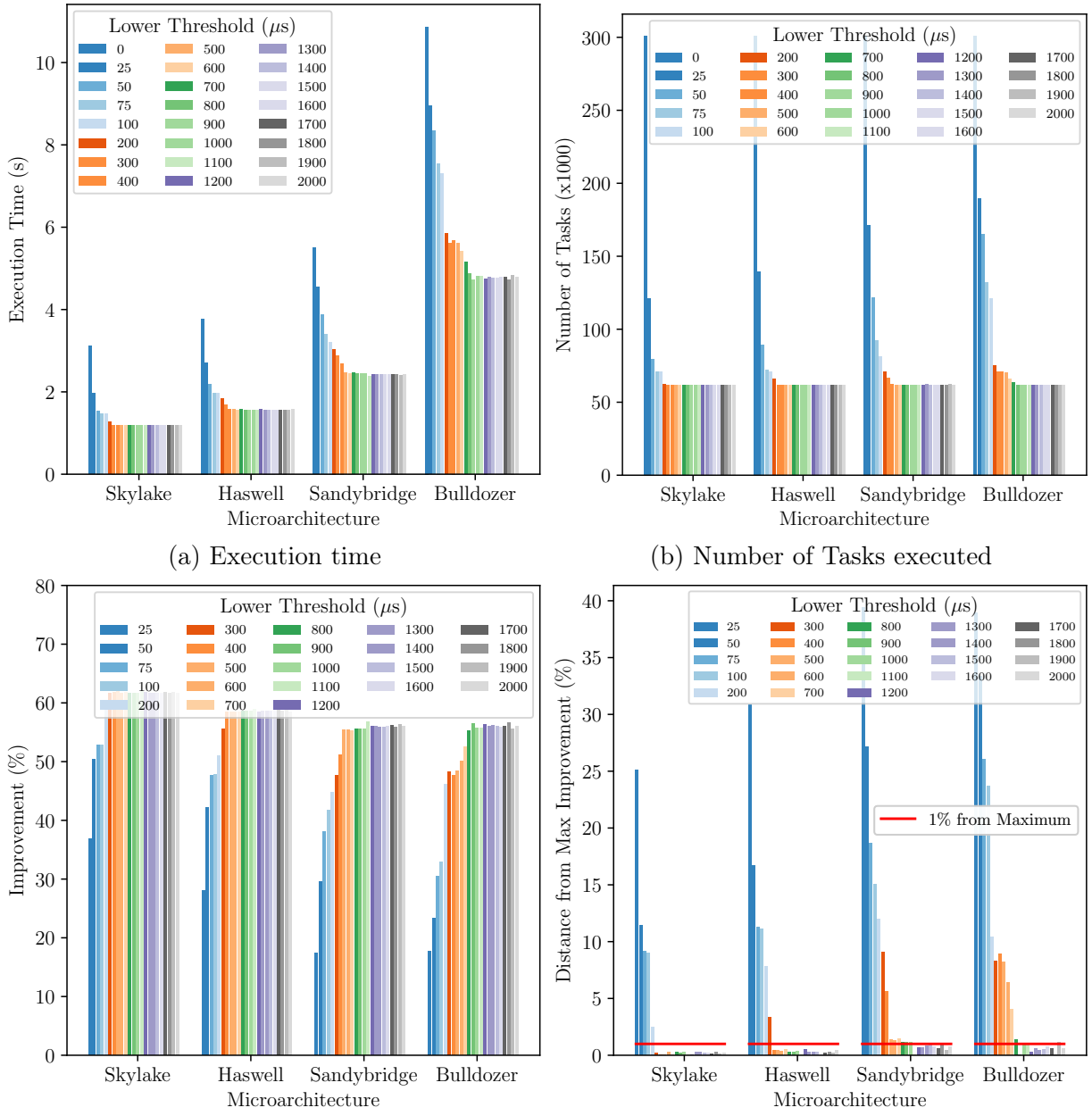

(a) Execution time

(b) Number of Tasks executed

(c) Improvement with respect to fully asynchronous execution

(d) Difference between current improvement and maximum improvement

Figure A.1. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Logistic Regression example running problem LRA-P1 on two threads. All values below the red line in (d) indicate those that are within one percent of maximum improvement.



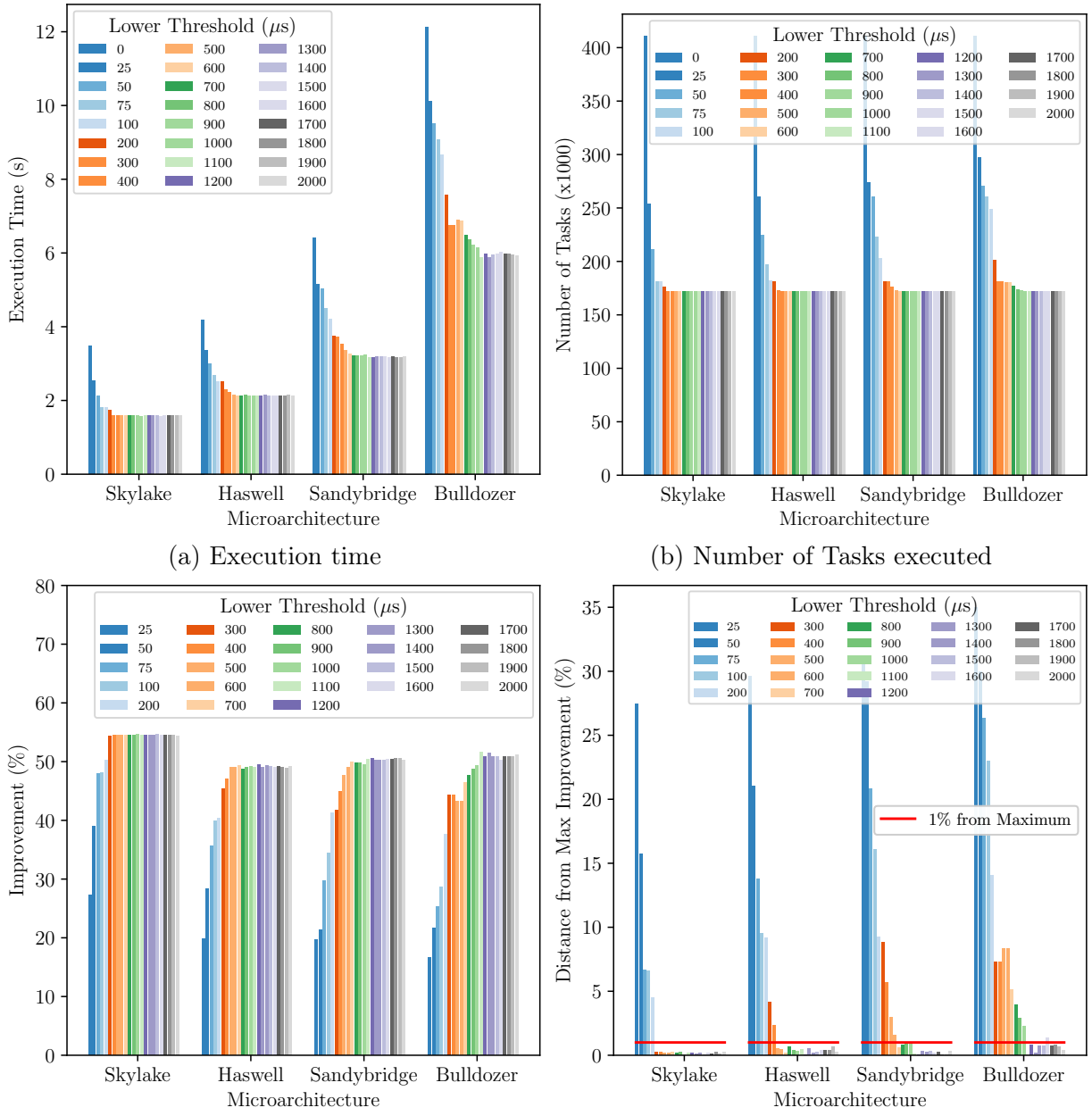

(a) Execution time

(b) Number of Tasks executed

(c) Improvement with respect to fully asynchronous execution

(d) Difference between current improvement and maximum improvement

Figure A.2. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Logistic Regression example running problem LRA-P1 on four threads. All values below the red line in (d) indicate those that are within one percent of maximum improvement.



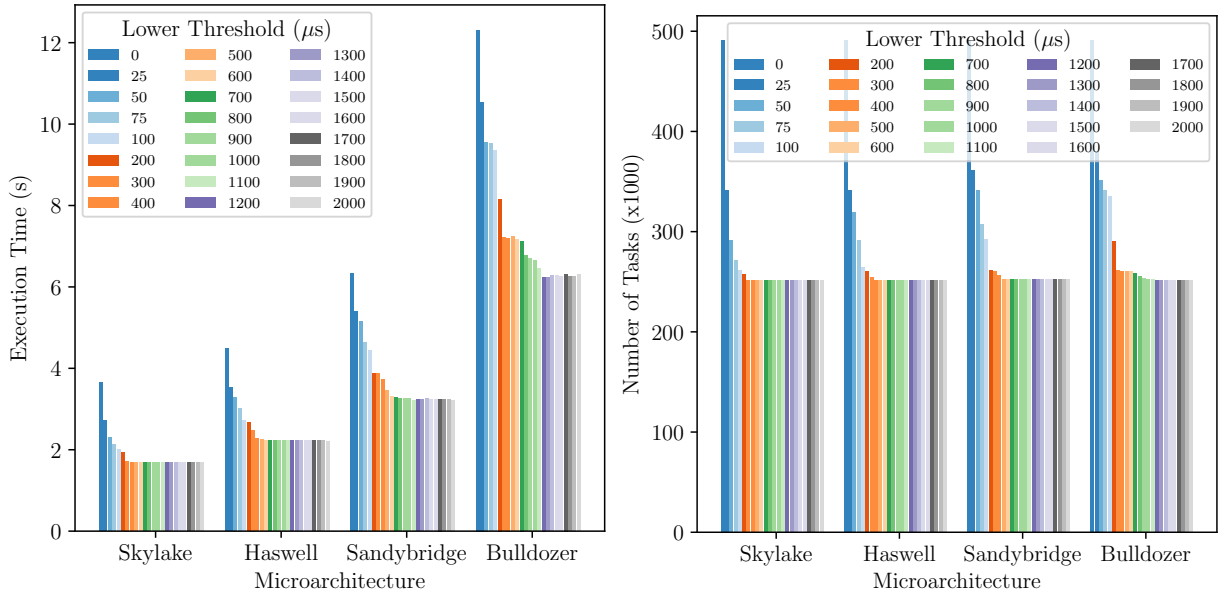

(a) Execution time

(b) Number of Tasks executed

(c) Improvement with respect to fully asynchronous execution

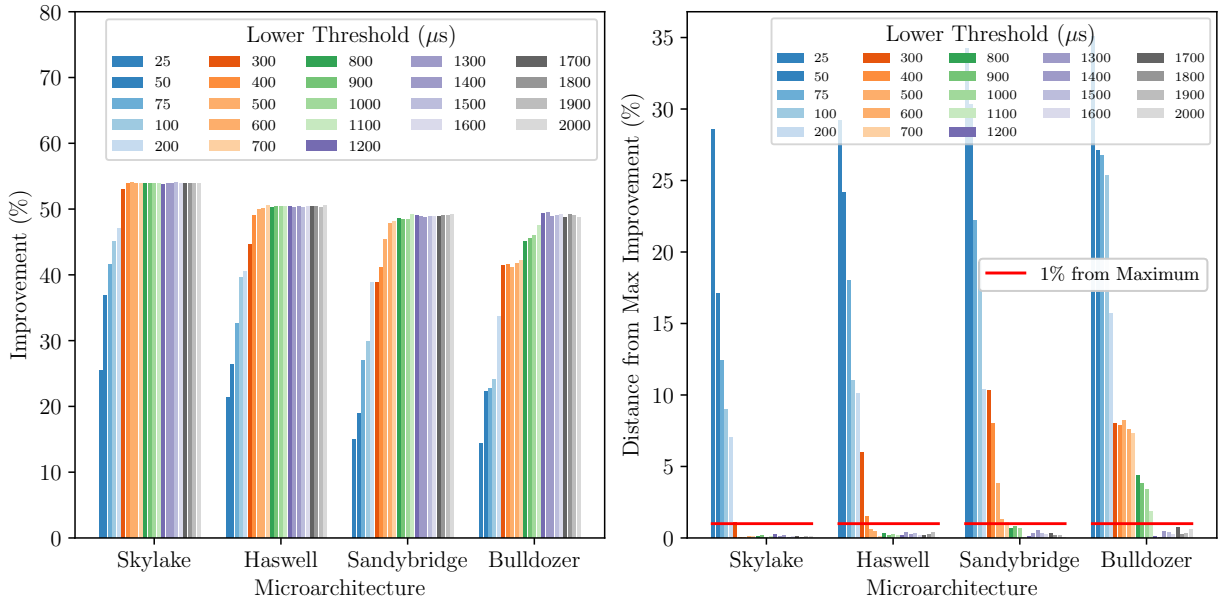
(d) Difference between current improvement and maximum improvement

Figure A.3. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Logistic Regression example running problem LRA-P2 on two threads. All values below the red line in (d) indicate those that are within one percent of maximum improvement.




(a) Execution time

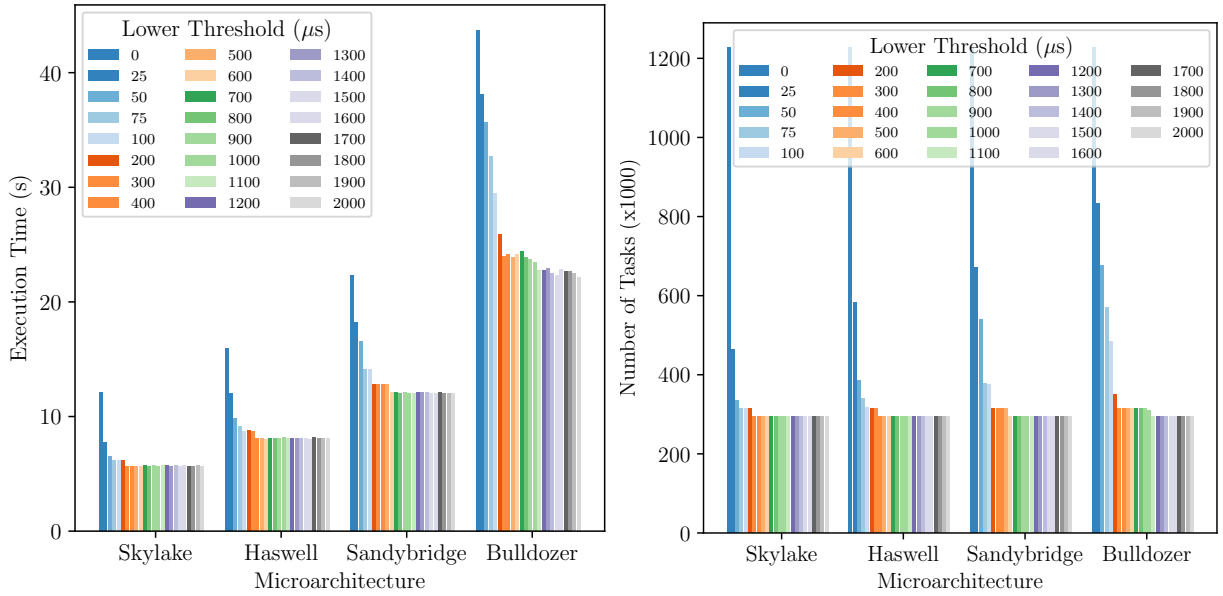
(b) Number of Tasks executed




(c) Improvement with respect to fully asynchronous execution

(d) Difference between current improvement and maximum improvement

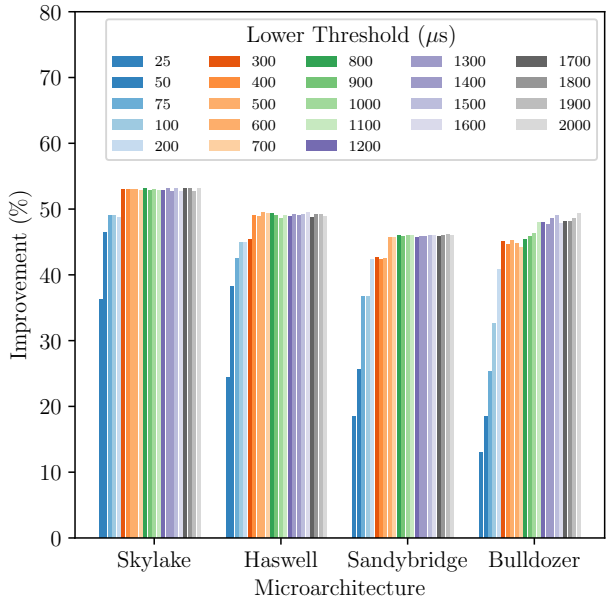
Figure A.4. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Logistic Regression example running problem LRA-P2 on four threads. All values below the red line in (d) indicate those that are within one percent of maximum improvement.



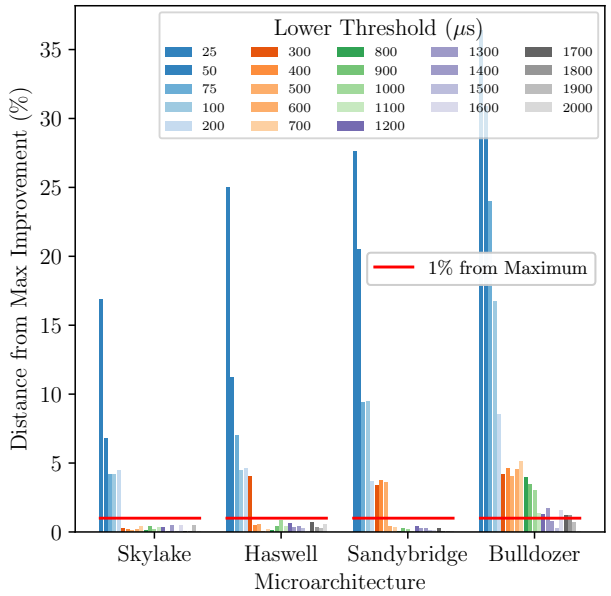

(a) Execution time




(b) Number of Tasks executed

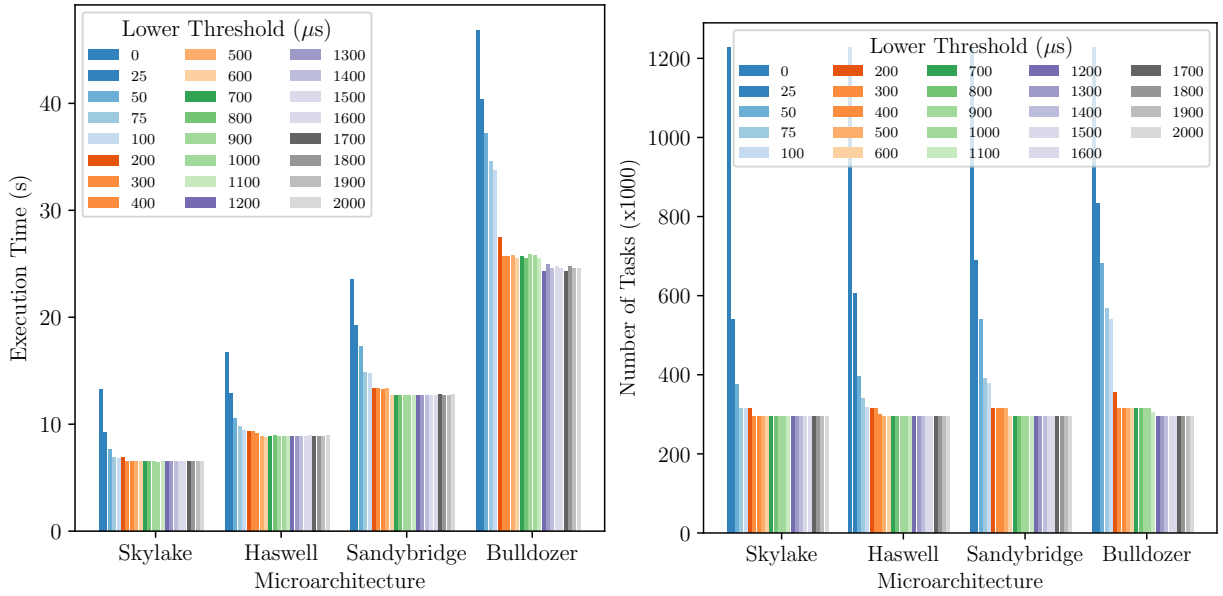



(c) Improvement with respect to fully asynchronous execution




(d) Difference between current improvement and maximum improvement

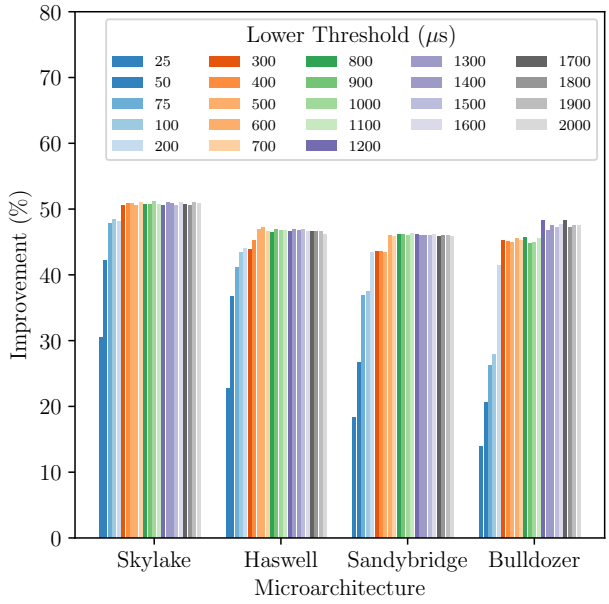
Figure A.5. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Alternating Least Squares example running problem ALS-P1 on two threads. All values below the red line in (d) indicate those that are within one percent of maximum improvement.



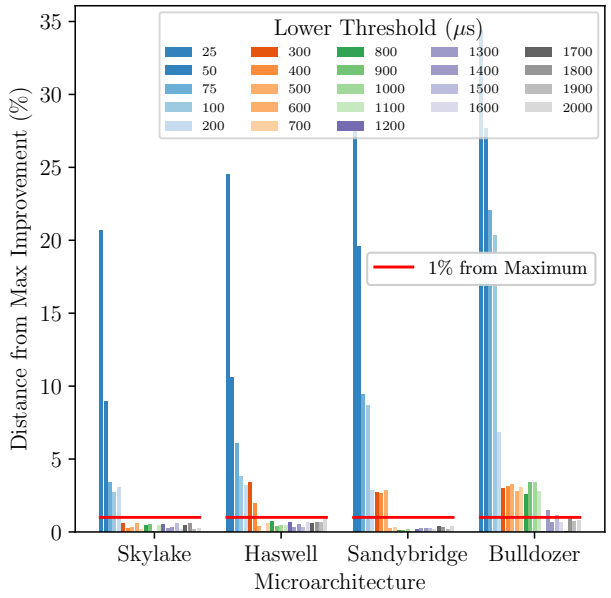

(a) Execution time




(b) Number of Tasks executed

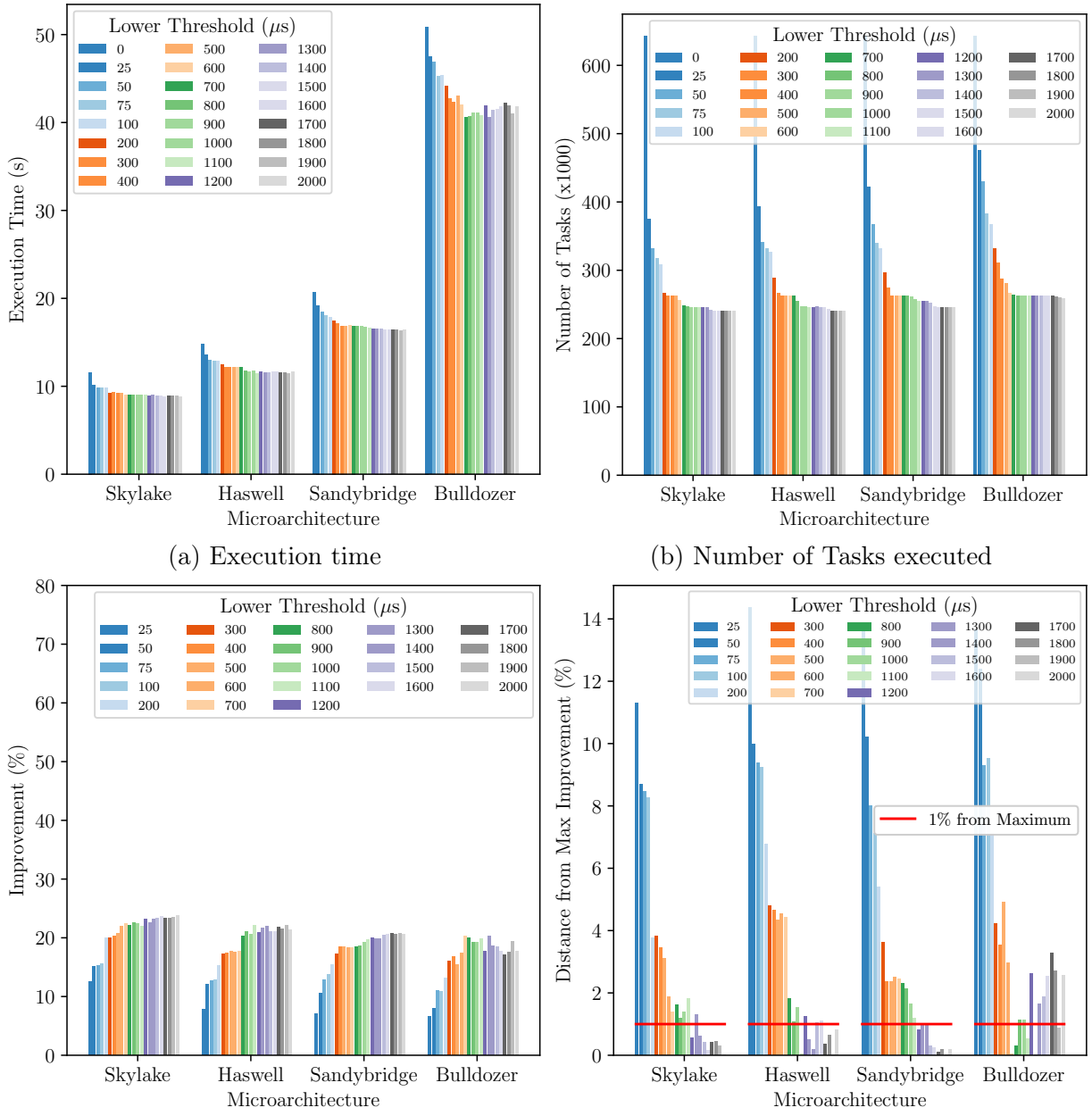



(c) Improvement with respect to fully asynchronous execution




(d) Difference between current improvement and maximum improvement

Figure A.6. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Alternating Least Squares example running problem ALS-P1 on four threads. All values below the red line in (d) indicate those that are within one percent of maximum improvement.



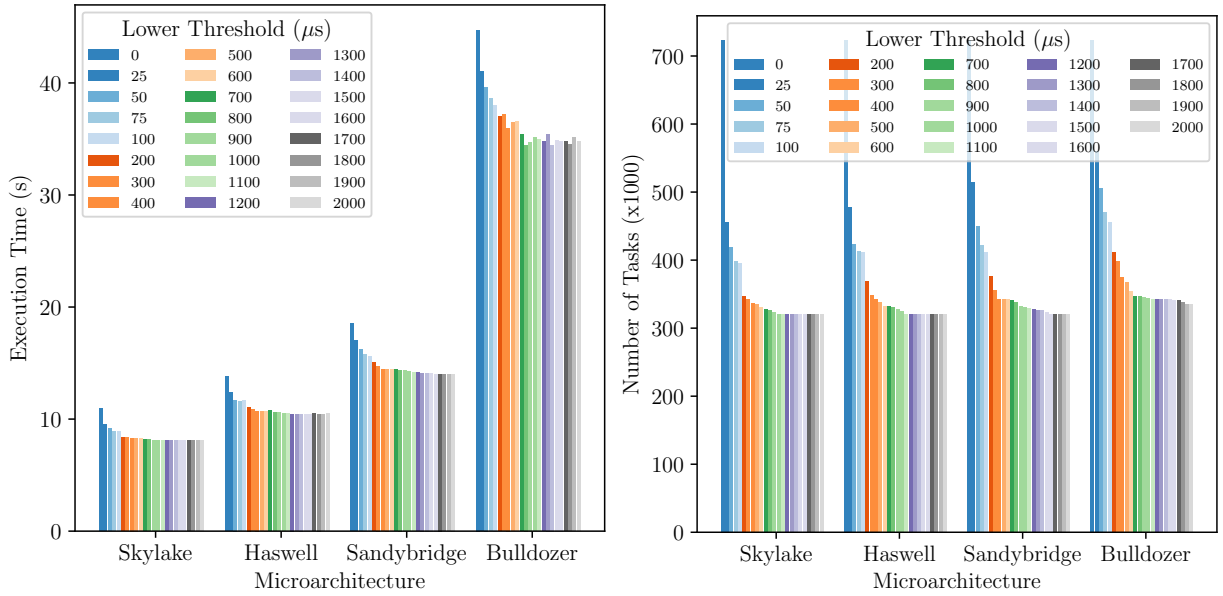

(a) Execution time

(b) Number of Tasks executed

(c) Improvement with respect to fully asynchronous execution

(d) Difference between current improvement and maximum improvement

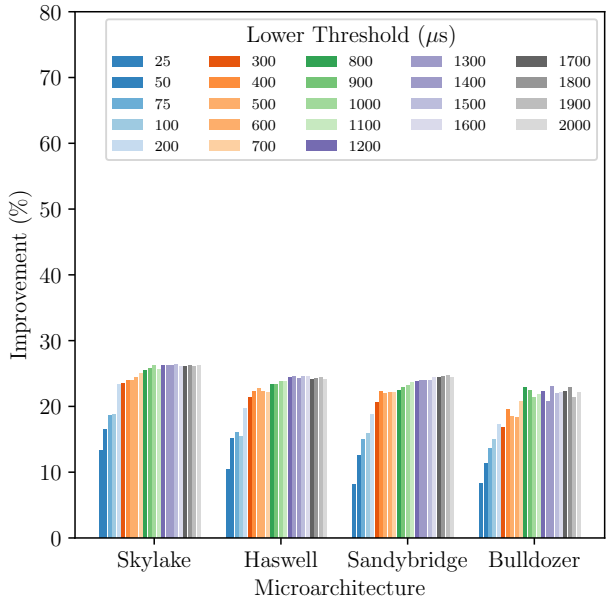
Figure A.7. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Alternating Least Squares example running problem ALS-P2 on two threads. All values below the red line in (d) indicate those that are within one percent of maximum improvement.



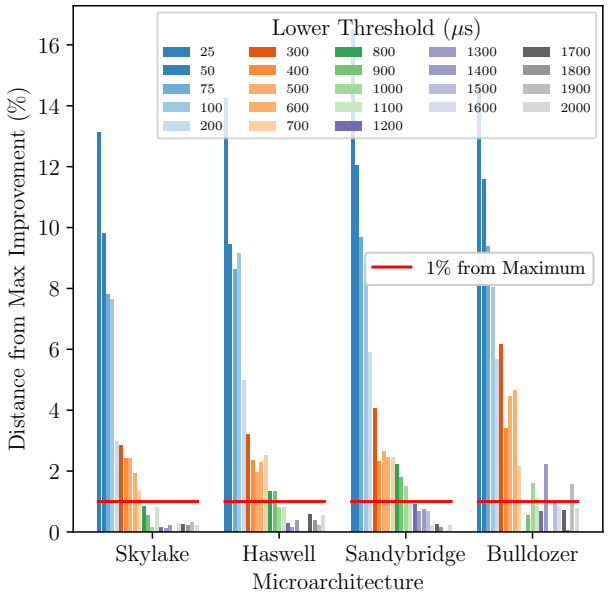

(a) Execution time




(b) Number of Tasks executed

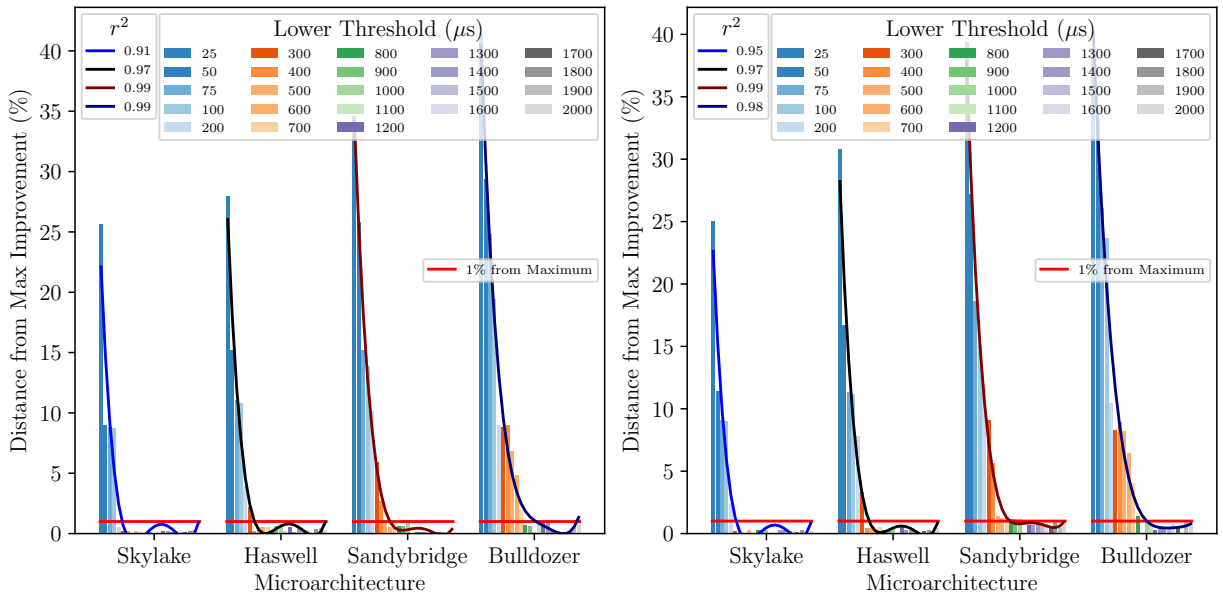



(c) Improvement with respect to fully asynchronous execution




(d) Difference between current improvement and maximum improvement

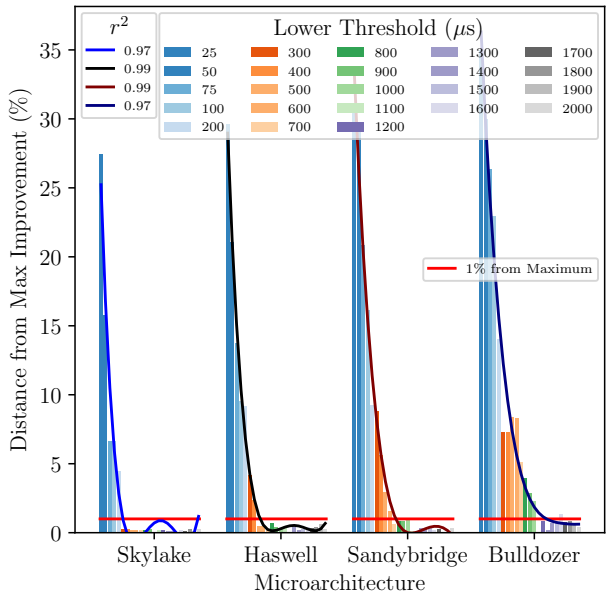
Figure A.8. (a) Execution time, (b) number of tasks executed, (c) improvement in application execution time compared to fully asynchronous case and (d) difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Alternating Least Squares example running problem ALS-P2 on four threads. All values below the red line in (d) indicate those that are within one percent of maximum improvement.



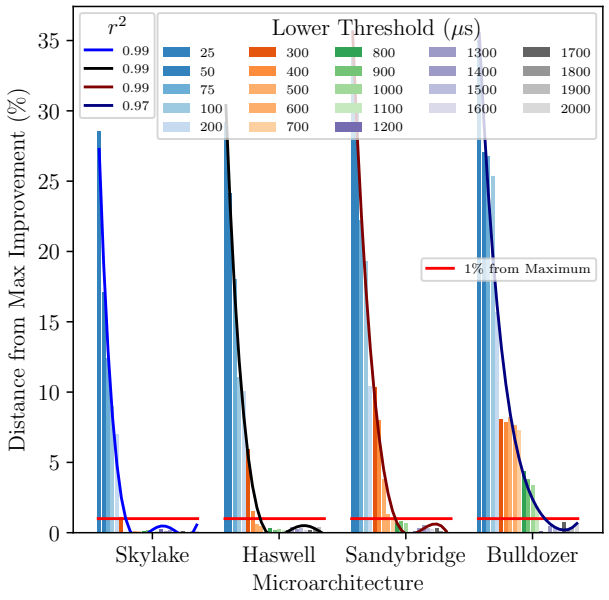

(a) LRA-P1 on two threads




(b) LRA-P1 on four threads

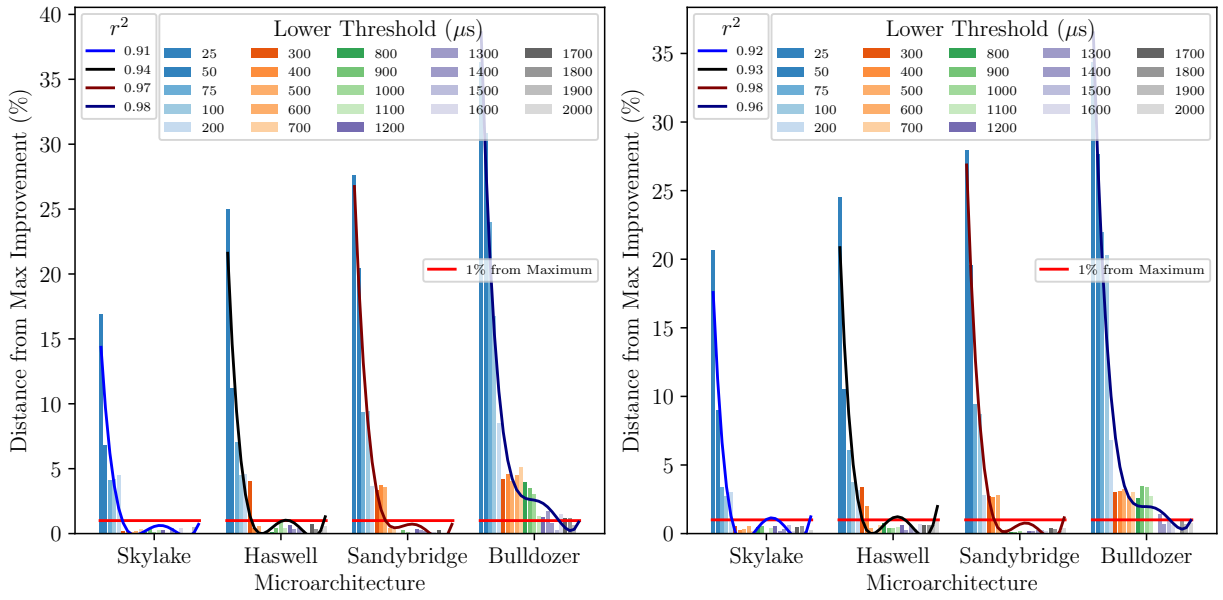



(c) LRA-P2 on two threads




(d) LRA-P2 on four threads

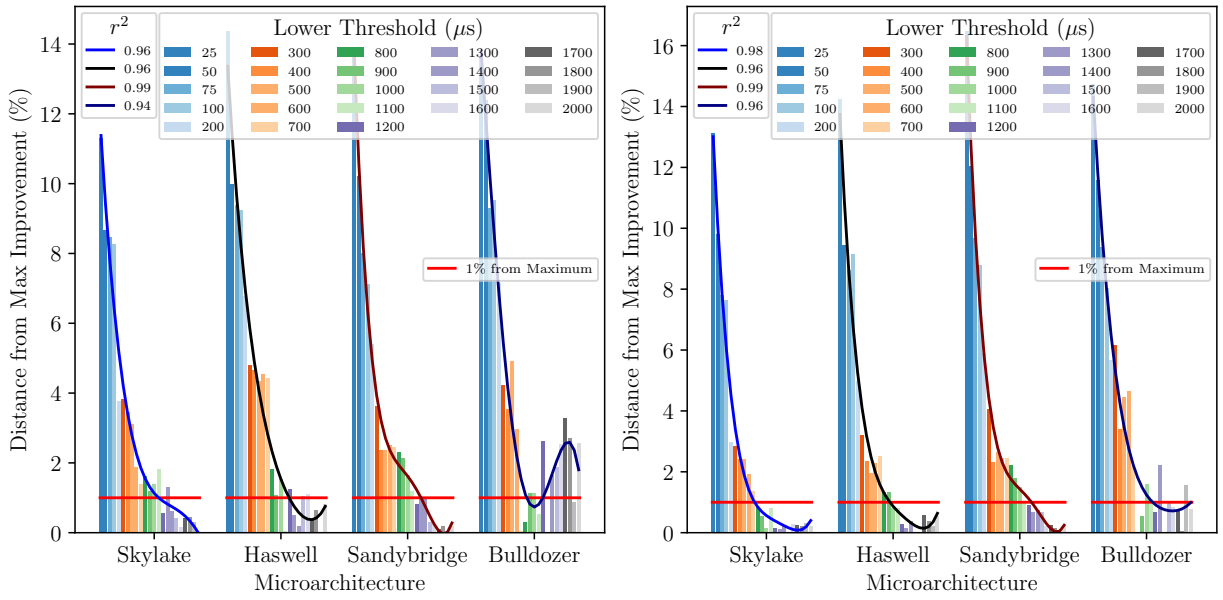
Figure A.9. Difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Logistic Regression example along with regression line. All values below the red line indicate those that are within one percent of maximum improvement.




(a) ALS-P1 on two threads




(b) ALS-P1 on four threads




(c) ALS-P2 on two threads




(d) ALS-P2 on four threads

Figure A.10. Difference between improvement using current threshold value and the threshold value that attains maximum improvement for the Alternating Least Squares example along with regression line. All values below the red line indicate those that are within one percent of maximum improvement.

# Appendix B. Copyright Information



**Title:** Methodology for Adaptive Active Message Coalescing in Task Based Runtime Systems

**Conference Proceedings:** 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

**Author:** Bibek Wagle

**Publisher:** IEEE

**Date:** May 2018





## Thesis / Dissertation Reuse

**The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:**

*Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:*

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

*Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:*

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

# ACM Information for Authors

**Author Rights** | FAQ

## ACM Author Rights

ACM  exists to support the needs of the computing community. For over sixty years ACM has developed publications and publication policies to maximize the visibility, impact, and reach of the research it publishes to a global community of researchers, educators, students, and practitioners. ACM has achieved its high impact, high quality, widely-read portfolio of publications with:

- Affordably priced publications
- Liberal Author rights policies
- Wide-spread, perpetual access to ACM publications via a leading-edge technology platform
- Sustainability of the good work of ACM that benefits the profession

### CHOOSE

Authors have the option to choose the level of rights management they prefer. ACM offers three different options for authors to manage the publication rights to their work.

- Authors who want ACM to manage the rights and permissions associated with their work, which includes defending against improper use by third parties, can use ACM's traditional copyright transfer agreement.
- Authors who prefer to retain copyright of their work can sign an exclusive licensing agreement, which gives ACM the right but not the obligation to defend the work against improper use by third parties.
- Authors who wish to retain all rights to their work can choose ACM's author-pays option, which allows for perpetual open access through the ACM Digital Library. Authors choosing the author-pays option can give ACM non-exclusive permission to publish, sign ACM's exclusive licensing agreement or sign ACM's traditional copyright transfer agreement. Those choosing to grant ACM a non-exclusive permission to publish may also choose to display a Creative Commons License on their works.

### POST

Authors can post the accepted, peer-reviewed version prepared by the author-known as the "pre-print"- to the following sites, with a DOI pointer to the Definitive Version of Record in the ACM Digital Library.

- On Author's own Home Page *and*
- On Author's Institutional Repository *and*
- In any repository legally mandated by the agency funding the research on which the work is based *and*

- On any non-commercial repository or aggregation that does not duplicate ACM tables of contents, i.e., whose patterns of links do not substantially duplicate an ACM-copyrighted volume or issue. Non-commercial repositories are here understood as repositories owned by non-profit organizations that do not charge a fee for accessing deposited articles and that do not sell advertising or otherwise profit from serving articles.

## DISTRIBUTE

Authors can post an *Author-Izer* link enabling free downloads of the Definitive Version of the work permanently maintained in the ACM Digital Library

- On the Author's own Home Page *or*
- In the Author's Institutional Repository.

## REUSE

Authors can reuse any portion of their own work in a new work of *their own* (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is *not* the editor, requires permission and usually a republication fee.

Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).

- Commercially produced course-packs that are *sold* to students require permission and possibly a fee.

## CREATE

ACM's copyright and publishing license include the right to make Derivative Works or new versions. For example, translations are "Derivative Works." By copyright or license, ACM may have its publications translated. However, ACM Authors continue to hold perpetual rights to revise their own works without seeking permission from ACM.

- If the revision is minor, i.e., less than 25% of new substantive material, then the work should still have ACM's publishing notice, DOI pointer to the Definitive Version, and be labeled a "Minor Revision of"
- If the revision is major, i.e., 25% or more of new substantive material, then ACM considers this a new work in which the author retains full copyright ownership (despite ACM's copyright or license in the original published article) and the author need only cite the work from which this new one is derived.

Minor Revisions and Updates to works already published in the ACM Digital Library are welcomed with the approval of the appropriate Editor-in-Chief or Program Chair.

## RETAIN

Authors retain all *perpetual rights* laid out in the ACM Author Rights and Publishing Policy, including, but not limited to:

- Sole ownership and control of third-party permissions to use for artistic images intended for exploitation in other contexts
- All patent and moral rights
- Ownership and control of third-party permissions to use of software published by ACM

Have more questions? Check out the FAQ.

back to top

## References


[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[2] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[3] Message Passing Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[4] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[5] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, Apr 2018.

[6] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex: An advanced parallel execution model for scaling-impaired applications. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW '09, pages 394–401, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[8] Vinay Chandra Amatya. *Parallel processes in HPX: designing an infrastructure for adaptive resource management.* PhD thesis, Louisiana State University, 2014.

[9] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[10] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[11] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[12] Thomas Heller. *Extending the C++ Asynchronous Programming Model with the HPX Runtime System for Distributed Memory Computing.* PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2019.

[13] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. *SIGARCH Comput. Archit. News*, 20(2):256–266, April 1992.

[14] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[15] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 362–376, Berlin, Heidelberg, 1974. Springer-Verlag.

[16] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2Nd Annual Symposium on Computer Architecture*, ISCA '75, pages 126–132, New York, NY, USA, 1975. ACM.

[17] Kevin Huck, Allan Porterfield, Nick Chaimov, Hartmut Kaiser, Allen Malony, Thomas Sterling, and Rob Fowler. An autonomic performance environment for exascale. *Supercomputing Frontiers and Innovations*, 2(3), 2015.

[18] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. Active pebbles: Parallel programming for data-driven applications. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 235–244, New York, NY, USA, 2011. ACM.

[19] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. Am++: A generalized active message framework. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 401–410, Sept 2010.

[20] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[21] B. Wagle, S. Kellar, A. Serio, and H. Kaiser. Methodology for adaptive active message coalescing in task based runtime systems. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1133–1140, May 2018.

[22] S. X. Yang, H. Fotso, J. Liu, T. A. Maier, K. Tomko, E. F. D'Azevedo, R. T. Scalettar, T. Pruschke, and M. Jarrell. Parquet approximation for the 4x4 Hubbard cluster. 80:046706, 2009.

[23] Patricia A Grubel. *Dynamic adaptation in HPX - A task-based parallel runtime system.* PhD thesis, New Mexico State University, 2016.

[24] Ste||ar Group. Running HPX on ROSTAM. `https://github.com/STEllAR-GROUP/hpx/wiki/Running-HPX-on-Rostam`, 2017.

[25] L. Wesolowski, R. Venkataraman, A. Gupta, J. S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale. Tram: Optimizing fine-grained communication with

topological routing and aggregation of messages. In *2014 43rd International Conference on Parallel Processing*, pages 211–220, Sept 2014.

[26] Yanhua Sun, Jonathan Lifflander, and Laxmikant V. Kalé. Pics: A performance-analysis-based introspective control system to steer parallel applications. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '14, pages 5:1–5:8, New York, NY, USA, 2014. ACM.

[27] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 185–197, New York, NY, USA, 1990. ACM.

[28] R. Tohid, B. Wagle, S. Shirzad, P. Diehl, A. Serio, A. Kheirkhahan, P. Amini, K. Williams, K. Isaacs, K. Huck, S. Brandt, and H. Kaiser. Asynchronous execution of python code on task-based runtime systems. In *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 37–45, Nov 2018.

[29] B. Wagle, M.A.H. Monil, K. Huck, A.D. Malony, A. Serio, and H. Kaiser. Runtime adaptive task inlining on asynchronous multitasking runtime systems. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, pages 76:1–76:10, New York, NY, USA, 2019. ACM.

[30] C Bishop. Pattern recognition and machine learning (information science and statistics), 1st edn. 2006. corr. 2nd printing edn. *Springer, New York*, 2006.

[31] Ste||ar Group. Phylanx - An Asynchronous Distributed C++ Array Processing Toolkit. `https://github.com/STEllAR-GROUP/phylanx/`, 2018.

[32] Olvi L Mangasarian and William H Wolberg. Cancer diagnosis via linear programming. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1990.

[33] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. Expression templates revisited: a performance analysis of current methodologies. *SIAM Journal on Scientific Computing*, 34(2):C42–C69, 2012.

[34] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 263–272. Ieee, 2008.

[35] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, December 2015.

[36] JD McCalpin. Stream: Sustainable memory bandwidth in high performance computers (2008), 1991-2007.

[37] OpenMP. The openmp api specification for parallel programming. Technical report, 2018.

[38] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[39] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[40] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[41] Chuck Pheatt. Intel® threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[42] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[43] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-t: A high-performance parallel lisp. *SIGPLAN Not.*, 24(7):81–90, June 1989.

[44] Alejandro Duran, Julita CorbalCorbalán, and Eduard Ayguadé. Evaluation of openmp task scheduling strategies. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *OpenMP in a New Era of Parallelism*, pages 100–110, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[45] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 36:1–36:11, Piscataway, NJ, USA, 2008. IEEE Press.

[46] Jianmin Bi, Xiaofei Liao, Yu Zhang, Chencheng Ye, Hai Jin, and Laurence T. Yang. An adaptive task granularity based scheduling for task-centric parallelism. In *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, HPCC '14, pages 165–172, Washington, DC, USA, 2014. IEEE Computer Society.

[47] Peter Thoman, Herbert Jordan, and Thomas Fahringer. Adaptive granularity control in task parallel programs using multiversioning. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 164–177, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[48] S. Iwasaki and K. Taura. A static cut-off for task parallel programs. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 139–150, Sep. 2016.

[49] S. Iwasaki and K. Taura. Autotuning of a cut-off for task parallel programs. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 353–360, Los Alamitos, CA, USA, sep 2016. IEEE Computer Society.

[50] D. Akhmetova, G. Kestor, R. Gioiosa, S. Markidis, and E. Laure. On the application task granularity and the interplay with the scheduling overhead in many-core shared memory systems. In *2015 IEEE International Conference on Cluster Computing*, pages 428–437, Sep. 2015.

[51] Yanhua Sun, Gengbin Zheng, Pritish Jetley, and Laxmikant V. Kalé. Parssse: an adaptive parallel state space search engine. *Parallel Processing Letters*, 21(3):319–338, 2011.

[52] Joshua Daniel Suetterlein. *A case for asynchronous many task runtimes: a modeling approach for high performance computing and Big Data analytics.* PhD thesis, University of Delaware, 2017.

[53] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[54] Joshua Landwehr, Joshua Suetterlein, Andrés Márquez, Joseph Manzano, and Guang R. Gao. Application characterization at scale: Lessons learned from developing a distributed open community runtime system for high performance computing. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 164–171, New York, NY, USA, 2016. ACM.

[55] C. D. Pham. Comparison of message aggregation strategies for parallel simulations on a high performance cluster. In *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728)*, pages 358–365, 2000.

## Vita

Bibek Wagle graduated with a Bachelors Degree in Engineering from Tribhuvan University, Kathmandu, Nepal in 2008. He completed his Master of Science degree from Teesside University, Middlesbrough, UK in 2011. He joined Louisiana State University for pursuing a doctoral degree in Computer Science in 2013. During his time at Louisiana State University, he worked with the STE||AR group and contributed to open source projects such as HPX and Phylanx.